aws

Entwicklerhandbuch

AWS Lambda



Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Lambda: Entwicklerhandbuch

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Die Marken und Handelsmarken von Amazon dürfen nicht in einer Weise in Verbindung mit nicht von Amazon stammenden Produkten oder Services verwendet werden, die geeignet ist, Kunden irrezuführen oder Amazon in irgendeiner Weise herabzusetzen oder zu diskreditieren. Alle anderen Marken, die nicht im Besitz von Amazon sind, gehören den jeweiligen Besitzern, die möglicherweise mit Amazon verbunden sind oder von Amazon gesponsert werden.

Table of Contents

Was ist AWS Lambda? ... 1
So vergleichen sich Lambda-Berechnungsprimitive ... 2
Anleitung zur Einrichtung von Agenten ... 5
Voraussetzungen ... 5
Claude Code ... 5
Installieren Sie das AWS-Core-Plugin aus dem Agent Toolkit für AWS ... 6
Codex ... 6
Installieren Sie das AWS-Core-Plugin aus dem Agent Toolkit für AWS ... 6
Cursor ... 6
Installieren AWS serverlose Fähigkeit ... 6
Kiro ... 6
Installieren AWS serverlose Fähigkeiten in Kiro CLI ... 6
Installieren Sie Kiro Powers (von Kiro IDE) ... 7
GitHub Copilot ... 7
Installieren AWS Fähigkeit ohne Server ... 7
Devin Desktop ... 7
Installieren AWS Fähigkeit ohne Server ... 7
OpenCode ... 7
Installieren AWS serverloser Skill ... 7
Für alle anderen Agenten, die mit Agent Skills kompatibel sind ... 7
Installieren AWS serverloser Skill ... 8
Zusätzliche Lambda-Skills installieren ... 8
Funktionen für Ereignisse ... 10
Wann sollten Event-Funktionen verwendet werden ... 10
Event-Funktionen und langlebige Funktionen ... 11
Funktionsweise von Lambda ... 12
Schlüssel-Features ... 12
Ähnliche Informationen ... 13
Schlüsselkonzepte ... 14
Lambda-Funktionen und -Funktionshandler ... 14
Lambda-Ausführungsumgebung und -Laufzeiten ... 15
Ereignisse und Auslöser ... 16
Lambda-Berechtigungen und -Rollen ... 17
Ausführen von Code ... 19

Erstellen ereignisgesteuerter Architekturen 36
Entwerfen einer Anwendung 49
Erstellen Sie Ihre erste Funktion 57
Melde dich an für AWS-Konto 57
Erstellen der Funktion 57
Aufrufen der -Funktion 64
Bereinigen 68
Nächste Schritte 69
Beispiel-Apps und -Muster 70
Dateiverarbeitung 70
Datenbankintegration 70
Geplante Aufgaben 71
Weitere Ressourcen 71
File-processing App 72
Scheduled-maintenance App 97
Programmiersprachen 118
Bauen mit Node.js 118
Bauen mit TypeScript 200
Erstellen mit Python 251
Erstellen mit Ruby 352
Erstellen mit Java 422
Erstellen mit Go 537
Erstellen mit C# 598
Bauen mit PowerShell 691
Erstellen mit Rust 706
Laufzeiten 733
Unterstützte Laufzeiten 734
Neue Laufzeit-Versionen 738
Richtlinie für den Laufzeitablauf 739
Modell der geteilten Verantwortung 740
Unterstützung der Verwendung nach der Einstellung 741
Empfangen von Benachrichtigungen über veraltete Laufzeitversionen 743
Veraltete Laufzeitumgebungen 745
Aktualisierungen der Laufzeitversion 749
Daten über Funktionen zur Laufzeit abrufen 762
Änderungen an der Laufzeitumgebung 769

Laufzeit-API ... 773
OS-only Laufzeiten ... 783
Open-Source-Repositorys ... 798
Konfiguration von Ereignisfunktionen ... 801
ZIP-Dateiarchive ... 804
Container-Images ... 818
Self-managed S3-Code-Speicher ... 828
Arbeitsspeicher ... 834
Flüchtiger Speicher ... 838
Befehlssätze (ARM/x86) ... 841
Zeitüberschreitung ... 845
Umgebungsvariablen ... 848
Funktionen an eine VPC anfügen ... 862
Funktionen an Ressourcen in einem anderen Konto anhängen ... 879
Internetzugriff für VPC-Funktionen ... 886
Eingehende Netzwerke ... 912
Dateisysteme ... 917
Aliase ... 926
Versionen ... 935
Tags (Markierungen) ... 940
Antwort-Streaming ... 945
Metadaten-Endpunkt ... 957
Aufrufen von Ereignisfunktionen ... 962
Synchroner Aufruf ... 964
Asynchroner Aufruf ... 968
Zuweisung von Ereignisquellen ... 984
Ereignisfilterung ... 997
Testen in der Konsole ... 1014
Funktionszustände ... 1018
Erneute Versuche ... 1022
Erkennung rekursiver Schleifen ... 1024
Funktions-URLs ... 1034
Skalierung von Eventfunktionen ... 1087
Verstehen und Visualisieren der Gleichzeitigkeit ... 1087
Berechnung der Parallelität für eine Event-Funktion ... 1092
Verständnis der reservierten Gleichzeitigkeit und der bereitgestellten Gleichzeitigkeit ... 1094

Gleichzeitigkeit und Anforderungen pro Sekunde verstehen 1104
Gleichzeitigkeitskontingente 1106
Konfigurieren reservierter Gleichzeitigkeit 1108
Konfigurieren von Provisioned Concurrency 1113
Skalierungsverhalten 1125
Überwachen der Gleichzeitigkeit 1127
Überwachung und Debugging 1134
Preisgestaltung 1134
Metriken zur Ereignisfunktion 1135
Protokolle der Ereignisfunktion 1148
CloudTrail protokolliert 1192
AWS X-Ray 1201
Einblicke in die Event-Funktion 1210
Anzeigen von Anwendungs-Metriken 1219
Anwendungssignale 1222
Debuggen mit VS Code 1226
Integration anderer Services 1229
Erstellen eines Auslösers 1230
Liste der Services 1230
Apache Kafka 1233
API Gateway 1407
Infrastructure Composer 1440
CloudFormation 1443
Amazon DocumentDB 1447
DynamoDB 1487
EC2 1552
Elastic Load Balancing (Application Load Balancer) 1554
Rufen Sie mit einem Scheduler auf EventBridge 1557
IoT 1560
Kinesis Data Streams 1562
Kubernetes 1626
MQ 1628
RDS 1651
S3 1683
Secrets Manager 1740
SQS 1760

Step Functions ... 1824
S3-Stapel ... 1835
SNS ... 1838
Dauerhafte Funktionen ... 1860
Wichtigste Vorteile ... 1861
Wann sollten langlebige Funktionen verwendet werden ... 1861
Funktionsweise ... 1863
Nächste Schritte ... 1864
Grundkonzepte ... 1865
Schaffung dauerhafter Funktionen ... 1869
Konfigurieren Sie langlebige Funktionen ... 1898
Dauerhafte Funktionen oder Step Functions ... 1901
Beispiele ... 1904
Sicherheit und Berechtigungen ... 1930
Verschlüsselung dauerhafter Ausführungsdaten ... 1937
Haltbares Ausführungs-SDK ... 1954
Dauerhafte Funktionen aufrufen ... 1964
Testen langlebiger Funktionen ... 1989
Überwachung langlebiger Funktionen ... 1991
Bewährte Methoden ... 1997
SnapStart ... 1998
Anwendungsfälle ... 2000
Unterstützte Funktionen und Einschränkungen ... 2000
Unterstützte Regionen ... 2001
Erwägungen zur Kompatibilität ... 2001
Preisgestaltung ... 2002
Aktiviert SnapStart ... 2004
Handhabung der Eindeutigkeit ... 2011
Laufzeit-Hooks ... 2018
Überwachen ... 2033
Sicherheitsmodell ... 2036
Bewährte Methoden ... 2037
Fehlerbehebung ... 2043
Isolierung von Mietern ... 2046
Wann sollte der Mandantenisolationsmodus verwendet werden ... 2047
Unterstützte Funktionen und Einschränkungen ... 2048

Unterstützt AWS-Regionen ... 2048
Überlegungen ... 2048
Preisgestaltung ... 2048
Isolationsmodus ... 2049
Aktivierung der Mandantenisolierung ... 2050
Funktionen mit Mandantenisolierung aufrufen ... 2053
Zugriff im Code ... 2058
Überwachen ... 2062
Fehlerbehebung ... 2064
Verwaltete Instances ... 2065
Wichtigste Funktionen ... 2065
Wann sollten Lambda Managed Instances verwendet werden ... 2065
Funktionsweise ... 2066
Modell der Parallelität ... 2066
Mietverhältnis und Isolierung ... 2067
Preisgestaltung ... 2067
Wie unterscheidet sich Lambda Managed Instances vom Lambda-Berechnungstyp (Standard) ... 2068
Nächste Schritte ... 2069
Erste Schritte ... 2069
Schlüsselkonzepte ... 2077
Bereitstellungen ... 2105
Laufzeiten ... 2114
Netzwerk ... 2140
Überwachen ... 2144
Kontingente ... 2162
Bewährte Methoden ... 2165
Fehlerbehebung ... 2170
Layer ... 2180
Verwenden von Ebenen ... 2182
Ebenen und Ebenenversionen ... 2183
Verpacken von Ebenen ... 2184
Erstellen und Löschen von Ebenen ... 2188
Hinzufügen von Ebenen ... 2192
Schichten mit CloudFormation ... 2196
Schichten mit AWS SAM ... 2197

Erweiterungen 2197
Ausführungsumgebung 2198
Auswirkungen auf Leistung und Ressourcen 2200
Berechtigungen 2200
Konfigurieren von Erweiterungen 2202
Partner für Erweiterungen 2204
Erweiterungs-API 2206
Telemetrie-API 2224
Fehlerbehebung 2283
Konfiguration 2284
Bereitstellung 2294
Aufruf 2301
Ausführung 2310
Zuordnung von Ereignisquellen 2321
Netzwerk 2327
Bewährte Methoden 2330
Code der Ereignisfunktion 2330
Konfiguration der Event-Funktion 2332
Skalierbarkeit der Event-Funktion 2333
Metriken und Alarme 2334
Arbeiten mit Streams 2335
Bewährte Methoden für die Gewährleistung der Sicherheit 2336
Beispielanwendungen 2337
Entwicklungstools 2340
Lokale Entwicklungstools 2341
Infrastructure as Code (IaC)-Tools 2341
GitHub Aktionen, Tools 2342
Powertools für AWS Lambda 2342
Tools für die Workflow- und Ereignisverwaltung 2343
Lokale Entwicklung 2343
GitHub Aktionen 2352
Infrastructure as Code (IaC) 2353
Elektrowerkzeuge für Lambda AWS 2378
Workflows und Ereignisse 2379
Testen 2382
Gezielte Geschäftsergebnisse 2384

Was muss getestet werden? ... 2384
So wird Serverless getestet ... 2385
Testmethoden ... 2385
Bewährte Methoden ... 2392
Herausforderungen beim Testen vor Ort ... 2396
Häufig gestellte Fragen ... 2398
Nächste Schritte und Ressourcen ... 2399
Berechtigungen ... 2400
Ausführungsrolle (Berechtigungen für Funktionen zum Zugriff auf andere Ressourcen) ... 2403
Zugriffsberechtigungen (Berechtigungen für andere Entitäten, auf Ihre Funktionen zuzugreifen) ... 2417
Web-Funktionen ... 2462
Wann sollten Webfunktionen verwendet werden ... 2463
Wie funktionieren Webfunktionen ... 2464
Wichtigste Vorteile ... 2464
Schlüsselkonzepte ... 2465
Ressourcenmodell ... 2465
Status der Revision ... 2467
Endpunktzustände ... 2468
Ausführungsumgebungen ... 2470
Bearbeitung von Anfragen ... 2471
Überarbeitungen ... 2472
Endpunkte ... 2483
Erstellen Sie Ihre erste Webfunktion ... 2490
Voraussetzungen ... 2490
Eine Webfunktion (Konsole) erstellen ... 2491
Eine Webfunktion erstellen (AWS CLI) ... 2493
Bereinigen ... 2498
Bereitstellung ... 2498
Kodieren Sie in Amazon S3 ... 2499
Bereitstellungsmuster ... 2505
Multi-Region Einsatz ... 2514
HTTP-Verhalten ... 2518
HTTP-Methoden ... 2519
Anfordern von Headern ... 2519
Antwort-Header ... 2520

Header-Grenzwerte ... 2521
Netzwerk ... 2521
Eingehende Konnektivität ... 2521
Ausgehende Konnektivität ... 2525
Eine Webfunktion mit Ihrer VPC verbinden ... 2526
VPC-Konnektivität aktualisieren oder entfernen ... 2530
IPv6-Support ... 2531
Skalierung ... 2532
Ausführungsumgebungen ... 2532
Dynamisches Parallelitätslimit pro Umgebung ... 2533
Einstellungen skalieren ... 2533
Kontingente für Konten ... 2535
Wie die Skalierungseinstellungen interagieren ... 2536
Drosselung ... 2538
Hochskalieren und Herunterskalieren ... 2538
Auswahl der Skalierungseinstellungen ... 2539
Skalierung für MultiRegion und PerRegion Endpunkte ... 2541
Beispiele für Skalierung ... 2541
Lebenszyklus der Umgebung ... 2546
Lebenszyklusphasen ... 2546
Initialisierung (Kaltstart) ... 2547
Scale-down und Herunterfahren ... 2548
Wiederverwendung der Umgebung ... 2549
Gleichzeitige Bearbeitung von Anfragen ... 2549
Laufzeiten ... 2550
Unterstützt Node.js Versionen ... 2550
Automatisches Runtime-Patching ... 2551
Unterstützte Web-Frameworks ... 2551
Node.js Ausführungsmodell ... 2552
Speicher und Node.js Haufen ... 2552
So leitet Lambda Anfragen an Ihre Anwendung weiter ... 2553
Konfiguration von Webfunktionen ... 2554
Bereitstellungspaket ... 2554
Architektur ... 2555
Zeitüberschreitung ... 2555
Umgebungsvariablen ... 2556

Tags (Markierungen) 2558
Metadaten-Endpunkt 2559
Berechtigungen 2561
Rolle „Ausführung“ 2562
Resource-based Richtlinien 2565
Berechtigungen für die Netzwerkkonfiguration 2572
Zugriffskontrolle 2573
Verwenden Sie den `IamAuth`Authentifizierungstyp 2574
Verwenden Sie den `ApplicationManaged`Authentifizierungstyp 2578
Governance und Zugriffskontrolle 2579
Verschlüsselung im Ruhezustand 2581
Optionen für die Verschlüsselung im Ruhezustand 2582
Verschlüsselung von Daten im Ruhezustand mithilfe eines vom Kunden verwalteten Schlüssels 2582
Überwachung der Interaktion der Webfunktionen mit AWS KMS 2593
Überwachen 2595
Abrechnung per Telemetrie 2596
Überlegungen zur Telemetrie 2597
Metriken 2599
Protokolle 2607
Benutzerdefinierte Metriken 2627
CloudTrail protokolliert 2630
Beispielanwendungen 2640
Minimal Express Anwendung 2641
Ehre API und statische Dateien 2641
Server-sent Ereignisse 2642
Multi-Region Route zur Überprüfung 2642
Ein Beispiel verpacken und bereitstellen 2643
Bewährte Methoden 2643
Schützen Sie jeden Endpunkt 2643
Behalten Sie unveränderliche Einrichtungsartefakte bei 2644
Design für gleichzeitige Anfragen 2645
Halten Sie den Start vorhersehbar 2645
Legen Sie anhand des gemessenen Verhaltens Grenzwerte fest 2646
Bereitstellung mit expliziter Überprüfung und Rollback 2646
Erstellen Sie eine operative Ausgangsbasis 2647

Betrieb MultiRegion und PerRegion Endgeräte als verteilte Systeme 2647
Testen 2648
Empfohlene Testschichten 2648
Führen Sie den HTTP-Server lokal aus 2649
Testen Sie das Bereitstellungspaket 2649
Testen Sie in einer isolierten Cloud-Umgebung 2650
Autorisierung und Offenlegung von Tests 2650
Testen Sie Parallelität, Skalierung und Drosselung 2651
Testbereitstellungen und Rollback 2651
Fehlerbehebung 2652
Sammeln von Diagnoseinformationen 2652
Probleme bei der Bereitstellung und beim Start 2653
CLI- und Paketfehler 2654
Fehler bei HTTP-Anfragen 2656
Grenzwerte für Header anfordern 2660
Fehlertypen der Webfunktion 2662
Protokolle oder Metriken fehlen 2665
Probleme mit Codespeicherung und Verschlüsselung 2666
Probleme mit und Endpunkten `MultiRegionPerRegion` 2667
Netzwerk- und Verbindungsprobleme 2667
Bereiten Sie ein Eskalationspaket vor 2669
Ähnliche Informationen 2669
MicroVMs 2670
Wie funktionieren Lambda MicroVMS 2670
Wann sollte Lambda MicroVMS verwendet werden 2671
Schlüssel-Features 2672
Schlüsselkonzepte 2672
Die wichtigsten Konzepte 2672
Wie Lambda Ihr Image erstellt 2673
MicroVM-Lebenszyklus 2674
Erstellen Sie Ihre erste MicroVM 2676
Voraussetzungen 2677
Erstellen Ihres ersten MicroVM-Images 2678
Führen Sie Ihre erste MicroVM aus 2680
Stellen Sie eine Verbindung zu Ihrer MicroVM her 2681
Bereinigen 2682

Nächste Schritte ... 2682
MicroVM-Images ... 2682
Wie Lambda ein MicroVM-Image erstellt ... 2683
Dimensionierung von MicroVM ... 2683
MicroVM-Basisimages ... 2684
Hooks zum Erstellen von MicroVM-Images ... 2685
Aktualisierung eines MicroVM-Images ... 2687
Image-Status und Build-Status ... 2688
Umgebungsvariablen ... 2689
-Snapshots ... 2689
MicroVM-Image-Patching ... 2691
Container-Basisimages ... 2691
Funktionen des Betriebssystems ... 2693
MicroVMS ausführen ... 2693
Eine MicroVM starten ... 2693
Verbindung zu einer MicroVM herstellen ... 2696
Lebenszyklus-Hooks ... 2699
Aussetzen und Fortsetzen von MicroVMs ... 2703
Skalierung und Parallelität ... 2704
Beenden einer MicroVM ... 2705
MicroVMs auflisten ... 2705
Fehlerbehandlung ... 2706
Netzwerk ... 2707
-Übersicht ... 2707
Eingehende Konnektivität ... 2707
Ausgehende Konnektivität ... 2711
Arbeiten mit ausgehenden Netzwerkverbindern ... 2712
Verwendung von Lambda MicroVMS mit Schnittstellen-VPC-Endpunkten (AWS PrivateLink) ... 2715
Sicherheit ... 2721
IAM-Rollen ... 2721
Authentifizierungstoken ... 2721
Shell-Zugriffstoken ... 2722
Taggen von Ressourcen ... 2722
Referenz zu IAM-Berechtigungen ... 2723
ARN-Formate für Ressourcen ... 2724

Beispiel: Least-privilege Richtlinie für einen Operator ... 2724
Überwachen ... 2725
Protokollierung (Protokolle) CloudWatch ... 2725
CloudTrail Protokollierung ... 2726
Fehlerbehebung ... 2730
Shell-Zugriff ... 2730
Fehlerbehebung ... 2731
Häufige Fehler (Image-Erstellung) ... 2732
Fehlerbehebung beim Netzwerkanschluss ... 2733
Bewährte Methoden ... 2735
Entwurf der Anwendung ... 2735
Leistung ... 2736
Sicherheit ... 2736
Kostenoptimierung ... 2736
Überwachen ... 2737
Integrationen ... 2737
Von Claude verwaltete Agenten ... 2738
Cursor Cloud-Agenten ... 2742
OpenAI Agenten-API ... 2748
Sicherheit, Governance und Compliance ... 2760
Datenschutz ... 2761
Verschlüsselung während der Übertragung ... 2762
Verschlüsselung im Ruhezustand ... 2763
Verwenden von servicegebundenen Rollen ... 2769
Service-linked Rollenberechtigungen für Lambda ... 2769
Erstellen einer dienstverknüpften Rolle für Lambda ... 2770
Bearbeitung einer serviceverknüpften Rolle für Lambda ... 2770
Löschen einer dienstverknüpften Rolle für Lambda ... 2770
Unterstützte Regionen für serviceverknüpfte Lambda-Rollen ... 2771
Identitäts- und Zugriffsverwaltung ... 2773
Zielgruppe ... 2773
Authentifizierung mit Identitäten ... 2774
Verwalten des Zugriffs mit Richtlinien ... 2775
Wie AWS Lambda funktioniert mit IAM ... 2777
Identity-based politische Beispiele ... 2783
AWS verwaltete Richtlinien ... 2786

Fehlerbehebung ... 2799
Governance ... 2801
Proaktive Kontrollen mit Guard ... 2804
Proaktive Kontrollen mit AWS Config ... 2808
Detektive Kontrollen mit AWS Config ... 2816
Codesignatur ... 2821
Scannen von Code ... 2824
Beobachtbarkeit ... 2829
Compliance-Validierung ... 2838
Ausfallsicherheit ... 2838
Sicherheit der Infrastruktur ... 2840
Sicherung von Workloads mit öffentlichen Endpunkten ... 2840
Authentifizierung und Autorisierung ... 2840
API-Endpunkte schützen ... 2841
Codesignatur ... 2842
Signaturvalidierung ... 2843
Konfiguration erstellen ... 2844
Berechtigungen ... 2846
Code-Signaturkonfigurations-Tags ... 2847
Arbeiten mit AWS SDKs ... 2852
Codebeispiele mit SDK ... 2854
Grundlagen ... 2857
Hallo Lambda ... 2858
Kennenlernen der Grundlagen ... 2867
Aktionen ... 3004
Szenarien ... 3142
Bestätigen Sie automatisch bekannte Benutzer mit einer Lambda-Funktion ... 3143
Migrieren Sie automatisch bekannte Benutzer mit einer Lambda-Funktion ... 3184
Erstellen Sie eine REST-API zum Verfolgen COVID-19 Daten ... 3207
Leihbibliothek-REST-API erstellen ... 3208
Erstellen einer Messenger-Anwendung ... 3209
Erstellen Sie eine Rest-API mit Funktions-Proxy-Integration ... 3210
Erstellen einer Serverless-Anwendung zur Verwaltung von Fotos ... 3219
Erstellen einer Websocket-Chat-Anwendung ... 3223
Erstellen einer Anwendung zum Analysieren von Kundenfeedback ... 3224
Erstellen eines Überwachungs-Dashboards mit einem Funktionsnamen als Variable ... 3230

Erstellen Sie Ihre erste serverlose Funktion 3240
Aufrufen einer Lambda-Funktion von einem Browser aus 3252
Transformieren von Daten mit S3 Object Lambda 3253
Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion 3254
Verwenden von Step Functions, um Lambda-Funktionen aufzurufen 3256
Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion 3257
Verwenden der Neptune-API, um Grafikdaten abzufragen 3259
Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen 3259
Schreiben von benutzerdefinierten Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito-Benutzerauthentifizierung 3268
Serverless-Beispiele 3291
Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion 3292
Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser 3311
Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser 3322
Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser 3331
Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser 3343
Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser 3353
Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen 3365
Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser 3374
Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser 3383
Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser 3397
Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser 3408
AWS Beiträge der Community 3418
Erstellen und Testen einer Serverless-Anwendung 3418
Lambda-Kontingente 3421
Datenverarbeitung und Speicherung 3423
Funktionskonfiguration, -bereitstellung und -ausführung 3424
Lambda-API-Anforderungen 3427
Dauerhafte Funktionen 3429
MicroVMs 3431
Datenverarbeitung und Speicherung 3431
Bilder und Versionen 3432
Per-MicroVM Durchsatz 3433

API-Ratenlimits 3433
Lambda-Web-Funktionen 3434
Kontingente für Webfunktionen, Revisionen und Ausführungsumgebungen 3435
Endpunktkontingente 3436
Kontenkontingente 3437
Netzwerkbeschränkungen 3438
Sonstige -Services 3438
Dokumentverlauf 3440
Frühere Updates 3473
..... mmmcdlxxxii

Was ist AWS Lambda?

AWS Lambda ist ein serverloser Rechendienst. Mit -Lambda können Sie ohne Server-Bereitstellung oder -Verwaltung Code ausführen. Lambda verwaltet automatisch die zugrunde liegende Infrastruktur — einschließlich Serverwartung, Kapazitätsbereitstellung, Skalierung und Patching —, sodass Sie sich auf Ihre Anwendungslogik konzentrieren können.

Lambda bietet drei Rechenprimitive, die jeweils für unterschiedliche Workload-Muster konzipiert sind:

- Lambda-Ereignisfunktionen — Führen Sie Code als Reaktion auf Ereignisse oder API-Aufrufe aus, ohne Server zu verwalten. Sie schreiben eine Handler-Funktion, verbinden sie mit einem Trigger (API Gateway, Amazon S3, Amazon SQS und mehr als 200 andere AWS Dienste) und Lambda führt sie aus. EventBridge Jeder Aufruf wird unabhängig ohne gemeinsamen Status ausgeführt und horizontal skaliert, um der Nachfrage gerecht zu werden. Lambda verwaltet Ausführungsumgebungen, Skalierung, Routing und Fehlertoleranz.
- Lambda Web Functions — Führen Sie Webanwendungen und API-Backends als HTTP-Server aus. Ihr Node.js Code überwacht einen Port und verarbeitet HTTP-Anfragen direkt mithilfe von Web-Frameworks wieExpress,Hono, oder React Router ohne Adapter. Jede Ausführungsumgebung bedient mehrere gleichzeitige Anfragen. Lambda bietet einen HTTPS-Endpunkt, skaliert Ihre Anwendung auf der Grundlage des Datenverkehrs (auch auf Null) und kann ihn überall replizieren. AWS-Regionen
- Lambda MicroVMS — Isolierte Rechenumgebungen mit nahezu sofortigem Start und einer Gesamtlebensdauer von bis zu 8 Stunden (einschließlich laufender und unterbrochener Zeit). Konzipiert für Workloads, die eine eigene Rechenumgebung für jeden einzelnen Benutzer oder Job benötigen. Lambda verwaltet Isolierung, Kapazität und Netzwerke. Ihre Anwendung verwendet Lambda MicroVMS-APIs und HTTPS-Endpunkte, um sie jeweils mit ihrer Rechenumgebung user/job zu verbinden.

> **Note**
>
> Event Functions ist der Name für den ursprünglichen Lambda-Funktionstyp, der Ihren Code als Reaktion auf Ereignisse und API-Aufrufe ausführt. Der Name unterscheidet diesen Funktionstyp von Lambda Web Functions und Lambda MicroVMS, und die Lambda-Konsole bezeichnet ihn als Event-Funktion. Die zugrunde liegende Ressource ist unverändert: Die API-Operationen (wie `CreateFunction` und`Invoke`), die AWS SDKs, die AWS CLI, der `AWS::Lambda::Function` Ressourcentyp und ARNs verwenden weiterhin die Funktion,

sodass Sie keinen Code, keine Infrastrukturvorlagen oder IAM-Berechtigungen ändern müssen.

Preisinformationen finden Sie unter AWS Lambda – Preise.

So vergleichen sich Lambda-Berechnungsprimitive

Lambda Event Functions, Lambda Web Functions und Lambda MicroVMS haben eine gemeinsame serverlose Grundlage:

- Kein Servermanagement — AWS verwaltet die zugrunde liegende Infrastruktur, das Instanz-Patching und die Kapazität.
- Pay-per-use Abrechnung — Keine Vorabverpflichtungen. Sie zahlen nur für die verwendeten Ressourcen.
- Verwaltetes Netzwerk — Alle drei bieten vom Service verwalteten eingehenden und ausgehenden Netzwerkzugriff.
- Firecracker-Virtualisierung — Isolierung zwischen Workloads. VM-level

Sie teilen sich zwar diese Grundlage, dienen aber unterschiedlichen Anwendungsfällen:

	Lambda-Ereignisfunktionen	Lambda-Webfunktionen	Lambda-MikroVMS
Am besten geeignet für	Request-response oder ereignisgesteuerte Workloads (APIs, Datenverarbeitung, Automatisierung)	Webanwendungen und API-Backends, die HTTP-Verkehr direkt bedienen	Persistente Umgebungen, in denen Benutzercode oder AI-produced nicht vertrauenswürdiger Code ausgeführt wird
Programmiermodell	Funktionshandler, der in einer unterstützten Laufzeit aufgerufen wurde	HTTP-Server, der auf einem Port lauscht und dabei Standard-Web-Frameworks verwendet	Beliebige Anwendung — führen Sie Ihre eigenen Binärdateien aus, überwachen Sie Ports, nutzen Sie die Funktione

	Lambda-Ereignisfunktionen	Lambda-Webfunktionen	Lambda-MikroVMS
			n des Linux-Betriebssystems
Duration (Dauer)	Standardmäßig bis zu 15 Minuten pro Aufruf; bis zu 90 Minuten pro asynchronem oder Event Source Mapping-Aufruf auf Lambda Managed Instances; mehrstufige Workflows mit einer Laufzeit von bis zu einem Jahr mit Lambda Durable Functions	Bis zu 15 Minuten pro HTTP-Anfrage (Standard 30 Sekunden)	Bis zu 8 Stunden pro Sitzung; sitzungsübergreifend aussetzen und fortsetzen
Laufzeitumgebung	Service-provided Sprachlaufzeiten; Unterstützung für vom Kunden bereitgestellte Laufzeiten	Service-provided Node.js-Laufzeiten	Customer-provided MicroVM-Images
Eingehende Netzwerke	Direkte Aufrufe oder Integrationen mit Diensten aus der Ereignisquelle AWS	Service-provided HTTPS-Endpunkt mit TLS; optionales latenzbasiertes Routing über AWS-Regionen	Eingehender Zugriff auf einen beliebigen Port mithilfe von OSI Layer 7-Protokollen
Concurrency (Nebenläufigkeit)	Standardmäßig jeweils eine Anfrage pro Ausführungsumgebung; mehrere gleichzeitige Anfragen pro Ausführungsumgebung für Lambda Managed Instances	Mehrere gleichzeitige Anfragen pro Ausführungsumgebung	Mehrere gleichzeitige Verbindungen pro MicroVM

	Lambda-Ereignisfunktionen	Lambda-Webfunktionen	Lambda-MikroVMS
Zustand der Umgebung	Ausführungsumgebungen können wiederverwendet werden (Warmstarts), aber der Status bleibt möglicherweise nicht über mehrere Aufrufe hinweg bestehen	Ausführungsumgebungen können wiederverwendet werden (Warmstarts), aber der Status bleibt möglicherweise nicht über mehrere Aufrufe hinweg bestehen	Speicher- und Festplattenstatus werden beim Anhalten beibehalten; beim Wiederaufnehmen wiederhergestellt
Skalierung	Automatisch — Lambda erstellt und zerstört Ausführungsumgebungen als Reaktion auf den Datenverkehr	Automatisch — Lambda erstellt und zerstört Ausführungsumgebungen als Reaktion auf den Datenverkehr und skaliert im Leerlauf auf Null	Developer-controlled — Sie erstellen MicroVMs über die API, setzen sie aus, setzen sie fort und beenden sie
Lebenszyklus	Vollständig von Lambda verwaltet	Vollständig von Lambda verwaltet	Developer-controlled; optionale Richtlinien für die automatische Unterbrechung und Wiederaufnahme im Leerlauf
Preise	Per-request + GB-seconds der Ausführungszeit	Pro Anfrage zuzüglich aktiver CPU-Zeit (in Schritten von 1 ms abgerechnet) und Arbeitsspeicher für die gesamte Lebensdauer jeder Ausführungsumgebung	Per-second Rechenleistung bei laufendem Betrieb + Snapshot-Speicher im Ruhezustand

Anleitung zur Agenteneinrichtung

KI-Codierungsagenten können die serverlose Entwicklung beschleunigen, indem sie intelligente Unterstützung für Anwendungsgerüst, Bereitstellungskonfiguration, Debugging und Infrastructure-as-Code-Generierung bieten. Indem Sie Ihren Agenten mit der richtigen Kombination aus Plugins und Fähigkeiten konfigurieren, statten Sie ihn mit dem Fachwissen aus, das für die Erstellung und Verwaltung serverloser Anwendungen erforderlich ist.

Arbeiten Sie mit Ihrem Coding-Agenten

Als Schnellstart-Referenz können Sie diesen Link direkt in den Kontext Ihres Agenten einfügen:

```
https://docs.aws.amazon.com/lambda/latest/dg/samples/aws-lambda-agent-setup.md
```

Wählen Sie die Installationsmethode, die zu Ihrer Entwicklungsumgebung passt.

Voraussetzungen

Stellen Sie sicher, dass die folgenden Voraussetzungen für die Installation des AWS-Core-Plug-ins erfüllt sind:

- uv ist auf Ihrem System installiert.
- (Optional) Ein AWS-Konto mit IAM-Anmeldeinformationen, die auf Ihrem lokalen Computer eingerichtet sind. Anmeldeinformationen sind für Tools erforderlich, die AWS API-Aufrufe ausführen und Skripts ausführen, jedoch nicht für die Suche nach Dokumentation oder das Auffinden von Fähigkeiten. Wenn Sie keine Anmeldeinformationen konfiguriert haben, finden Sie detaillierte Anweisungen unter Einrichten des AWS MCP-Servers.

Claude Code

Installieren Sie das AWS-Core-Plug-In, das Teil des Agent-Toolkits für AWS ist und die AWS MCP-Serverkonfiguration und ein kuratiertes Set an Agentenfähigkeiten (einschließlich des serverlosen Skills) bündelt. AWS

Installieren Sie das AWS-Core-Plugin aus dem Agent Toolkit für AWS

Führen Sie den folgenden Befehl aus, um das Plugin zu installieren:

```
/plugin marketplace add aws/agent-toolkit-for-aws
/plugin install aws-core@agent-toolkit-for-aws
```

Dadurch werden die Agentenfähigkeiten, einschließlich des AWS Serverless-Skills, der Agent-Hooks und der AWS MCP-Serverkonfiguration, in einem Schritt installiert.

Codex

Installieren Sie das AWS-Core-Plugin aus dem Agent Toolkit für AWS

Führen Sie den folgenden Befehl aus, um das Plugin zu installieren:

```
codex plugin marketplace add aws/agent-toolkit-for-aws
```

Starten Sie dann Codex und starten Sie es`/plugins`, um das AWS-Core-Plugin zu durchsuchen und zu installieren.

Cursor

Installieren AWS serverlose Fähigkeit

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Kiro

Installieren AWS serverlose Fähigkeiten in Kiro CLI

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Installieren Sie Kiro Powers (von Kiro IDE)

Installieren Sie die folgenden Kiro-Funktionen, die den Kiro-Agenten auf Abruf spezielle Kontexte und Tools zur Verfügung stellen:

- Serverloses Anwendungsmodell (SAM) — Installation über Kiro Powers Link.
- Lambda Managed Instances — Installation über Kiro Powers Link.

GitHub Copilot

Installieren AWS Fähigkeit ohne Server

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Devin Desktop

Installieren AWS Fähigkeit ohne Server

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

OpenCode

Installieren AWS serverloser Skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Für alle anderen Agenten, die mit Agent Skills kompatibel sind

Gehen Sie für jeden anderen Agenten, der das Open-Source-Format für Agentenfähigkeiten unterstützt, wie folgt vor:

Installieren AWS serverloser Skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Zusätzliche Lambda-Skills installieren

Das Agent Toolkit für AWS umfasst spezielle serverlose Fähigkeiten, die Ihren KI-Agenten Wissen über Lambda-Funktionen vermitteln. Installieren Sie diese Skills mit einem beliebigen Agenten wie Claude Code, Cursor, Kiro CLI, GitHub Copilot, Codex, Devin Desktop OpenCode oder anderen Agenten, die mit dem Open-Source-Agenten-Skills-Format kompatibel sind, damit Ihr Agent Sie beim Erstellen, Bereitstellen und Verwalten von Workloads mithilfe von Lambda Web Functions, Lambda MicroVMS, Lambda Managed Instances und Lambda Durable Functions unterstützen kann.

1. Lambda Web Functions Skill — Mit Lambda Web Functions können Sie Webanwendungen und API-Backends als HTTP-Server ausführen, wobei Ihr Node.js Code Anfragen direkt mithilfe von Frameworks wieExpress, oder verarbeitet FastifyHono, wobei Lambda einen HTTPS-Endpunkt, automatische Skalierung (auch auf Null) und Replikation in mehreren Regionen bereitstellt.

   ```
   npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
   specialized-skills/serverless-skills/aws-lambda-web-functions --yes --global
   ```

2. Lambda MicroVMS-Kenntnisse — Lambda MicroVMS ist ein serverloses Compute-Primitiv, das VM-level Isolierung, nahezu sofortige Start- und Wiederaufnahmegeschwindigkeiten sowie Statuserhaltung für die Ausführung von Benutzern oder Code in isolierten, statusbehafteten Ausführungsumgebungen bietet. AI-generated

   ```
   npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
   specialized-skills/serverless-skills/aws-lambda-microvms --yes --global
   ```

3. Fähigkeiten mit Lambda Managed Instances — Mit Lambda Managed Instances können Sie Lambda-Funktionen auf Ihren Amazon EC2 EC2-Instances ausführen und gleichzeitig die einfache Bedienung von Lambda beibehalten. Sie erhalten Zugriff auf spezielle Rechenkonfigurationen und Kosteneffizienz durch Amazon EC2 EC2-Preisvorteile, ohne die Infrastruktur verwalten zu müssen.

   ```
   npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
   specialized-skills/serverless-skills/aws-lambda-managed-instances --yes --global
   ```

4. Fähigkeit zu dauerhaften Lambda-Funktionen — Mit dauerhaften Lambda-Funktionen können Sie fehlertolerante, lang laufende Anwendungen erstellen, die den Fortschritt überprüfen, sich automatisch nach Ausfällen wiederherstellen und die Ausführung für bis zu einem Jahr aussetzen können, wenn sie auf lang andauernde Aufgaben wie Human-in-the-Loop-Prozesse warten.

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/aws-lambda-durable-functions --yes --global
```

AWS Lambda Event-Funktionen (früher Lambda-Funktionen)

> **Tip**
>
> Nehmen Sie an kostenlosen praktischen Workshops von Experten für serverlose Lösungen teil, um zu erfahren, wie Sie mithilfe von Best Practices serverlose Anwendungen erstellen können. Melden Sie sich auf der Experience-Website für kostenlose serverlose Workshops an. AWS

AWS Lambda Event Functions, früher Lambda-Funktionen genannt, führen Ihren Code als Reaktion auf Ereignisse aus Quellen wie Amazon Simple Storage Service (Amazon S3), Amazon Simple Queue Service (Amazon SQS) aus EventBridge, ohne dass Server verwaltet werden müssen. Eine Funktion empfängt ein Ereignis, führt Ihren Handler aus und gibt eine Antwort zurück. Die Funktionen werden automatisch nach oben und unten skaliert, wobei die Preise pro Nutzung berechnet werden. Um zu beginnen, sehen Sie sich Erstellen Sie Ihre erste Funktion an. Informationen zu bewährten Methoden, Programmiersprachen, Mandantenisolierung und Tools für Entwicklung und Bereitstellung finden Sie in den entsprechenden Themen weiter unten.

> **Note**
>
> Event Functions ist der Name für den ursprünglichen Lambda-Funktionstyp, und die Lambda-Konsole bezeichnet ihn als Event-Funktion. Die zugrunde liegende Ressource ist unverändert: Die API-Operationen (wie `CreateFunction` und`Invoke`), die AWS SDKs, die AWS CLI und der `AWS::Lambda::Function` Ressourcentyp verwenden weiterhin Funktionen, sodass Sie keinen Code oder keine Konfiguration ändern müssen.

Preisinformationen finden Sie unter AWS Lambda – Preise.

Wann sollten Event-Funktionen verwendet werden

Sie können Ereignisfunktionen verwenden für:

- Dateiverarbeitung: Dateien werden automatisch verarbeitet, wenn sie auf Amazon Simple Storage Service hochgeladen werden. Einzelheiten finden Sie in den Beispielen zur Dateiverarbeitung.

- Long-running Workflows: Erstellen Sie statusbehaftete, mehrstufige Workflows, die bis zu einem Jahr lang ausgeführt werden können. Einzelheiten finden Sie unter langlebige Lambda-Funktionen.
- Beispiele für Datenbankoperationen und -integration: Reagieren Sie auf Datenbankänderungen und automatisieren Sie Datenworkflows. Einzelheiten finden Sie in den Datenbankbeispielen.
- Geplante und regelmäßige Aufgaben: Führen Sie automatisierte Abläufe regelmäßig aus mit EventBridge. Einzelheiten finden Sie in den Beispielen für geplante Aufgaben.
- Stream-Verarbeitung: Verarbeiten Sie Echtzeit-Datenströme für Analytik und Überwachung. Einzelheiten finden Sie unter Kinesis Data Streams.
- Webanwendungen: Verarbeiten Sie Web- und API-Anfragen über Amazon API Gateway, Application Load Balancer oder Lambda-Funktions-URLs. Standardmäßig verarbeitet jede Ausführungsumgebung jeweils eine Anfrage. Auf Lambda Managed Instances verarbeitet eine Ausführungsumgebung mehrere Anfragen gleichzeitig.
- Mobile Backends: Erstellen Sie sichere API-Backends für Mobil- und Webanwendungen.
- IoT-Backends: Verarbeiten Sie Web-, Mobil-, IoT- und Drittanbieter-API-Anfragen. Weitere Details finden Sie unter IoT.

Verwenden Sie Lambda Web Functions, um HTTP-Verkehr direkt von einem Standard-Webserver aus bereitzustellen. Weitere Informationen finden Sie unter Web-Funktionen.

Event-Funktionen und langlebige Funktionen

Lambda-Funktionen werden standardmäßig bis zu 15 Minuten pro Aufruf ausgeführt und eignen sich ideal für ereignisgesteuerte Aufgaben wie die Verarbeitung von API-Anfragen, die Bearbeitung von Datei-Uploads oder die Reaktion auf Datenbankänderungen. Auf Lambda Managed Instances können Funktionen für asynchrone Aufrufe und ESM-Aufrufe (Event Source Mapping) bis zu 90 Minuten lang ausgeführt werden. Dauerhafte Funktionen erweitern dieses Modell auf Workloads, die länger laufen und Unterbrechungen überstehen müssen. Sie überprüfen ihren Fortschritt automatisch, sodass sie nach Ausfällen zuverlässig wieder aufgenommen werden können. Sie können bis zu einem Jahr lang ausgeführt werden, wenn sie asynchron aufgerufen werden. Verwenden Sie langlebige Funktionen, wenn Sie mehrstufige Workflows, Genehmigungen durch Mitarbeiter oder die Koordination zwischen Diensten über längere Zeiträume benötigen.

Funktionsweise von Lambda

Wenn Sie Lambda verwenden, sind Sie nur für Ihren Code verantwortlich. Lambda führt Ihren Code auf einer hochverfügbaren Datenverarbeitungsinfrastruktur aus und verwaltet alle Datenverarbeitungsressourcen, einschließlich Server- und Betriebssystemwartung, Kapazitätsbereitstellung, automatische Skalierung und Protokollierung.

Da Lambda ein serverloser, ereignisgesteuerter Rechendienst ist, verwendet er einen anderen Programmieransatz als herkömmliche Webanwendungen. Das folgende Modell veranschaulicht die Funktionsweise von Lambda:

1. Sie schreiben und organisieren Ihren Code in Lambda-Funktionen, die die grundlegenden Bausteine für die Erstellung einer Lambda-Anwendung darstellen.
2. Sie kontrollieren Sicherheit und Zugriff über Lambda-Berechtigungen und verwalten mithilfe von Ausführungsrollen, mit welchen AWS Diensten Ihre Funktionen interagieren können und welche Ressourcenrichtlinien mit Ihrem Code interagieren können.
3. Ereignisquellen und AWS Dienste lösen Ihre Lambda-Funktionen aus, indem sie Ereignisdaten im JSON-Format übergeben, die von Ihren Funktionen verarbeitet werden (dazu gehören auch Zuordnungen von Ereignisquellen).
4. Lambda führt Ihren Code mit sprachspezifischen Laufzeiten (wie Python) in Ausführungsumgebungen aus, die Ihre Laufzeit, Ebenen Node.js und Erweiterungen verpacken.

> **Tip**
>
> Weitere Informationen zum Erstellen von Serverless-Lösungen finden Sie im Serverless-Benutzerhandbuch.

Schlüssel-Features

Konfigurieren, Steuern und Bereitstellen sicherer Anwendungen:

- Umgebungsvariablen: Modifizieren Sie das Verhalten von Anwendungen, ohne dass neuer Code bereitgestellt werden muss.
- Versionen: Testen Sie neue Funktionen auf sichere Weise und sorgen Sie gleichzeitig für stabile Produktionsumgebungen.

- Layer: Optimieren Sie die Wiederverwendung und Wartung von Code, indem Sie gemeinsame Komponenten über mehrere Funktionen hinweg gemeinsam nutzen.
- Codesignatur: Sorgen Sie für die Einhaltung der Sicherheitsbestimmungen, indem Sie sicherstellen, dass nur genehmigter Code die Produktionssysteme erreicht.

Zuverlässiges Skalieren Arbeiten:

- Parallelitäts- und Skalierungskontrollen steuern präzise die Reaktionsfähigkeit von Anwendungen und die Ressourcennutzung während Verkehrsspitzen.
- SnapStart: Reduzieren Sie die Kaltstartzeiten erheblich. Lambda SnapStart kann eine Startleistung von nur unter einer Sekunde bieten, in der Regel ohne Änderungen an Ihrem Event-Funktionscode.
- Antwort-Streaming: Optimieren Sie die Funktionsleistung, indem Sie große Nutzdaten schrittweise für die Echtzeitverarbeitung bereitstellen.
- Container-Images: Verpacken Sie Funktionen mit komplexen Abhängigkeiten mithilfe von Container-Workflows.

Nahtloses Verbinden und Integrieren:

- VPC-Netzwerke sichern sensible Ressourcen und interne Dienste.
- Die Dateisysteme-Integration nutzt persistente Daten gemeinsam und verwaltet zustandsbehaftete Vorgänge über Funktionsaufrufe hinweg.
- Funktions-URLs: Erstellen Sie öffentlich zugängliche APIs und Endpunkte ohne zusätzliche Dienste.
- Erweiterungen: Erweitern Sie die Funktionen durch Überwachungs-, Sicherheits- und Betriebstools.

Ähnliche Informationen

- Weitere Informationen zur Funktionsweise von Lambda finden Sie unter Funktionsweise von Lambda.
- Informationen zum Einstieg in Lambda finden Sie unter Erstellen Sie Ihre erste Lambda-Funktion.
- Eine Liste von Beispielanwendungen finden Sie unter Erste Schritte mit Beispielanwendungen und -mustern.

Funktionsweise von Lambda

Lambda-Funktionen sind die grundlegenden Bausteine, die Sie zum Erstellen von Lambda-Anwendungen verwenden. Beim Schreiben von Funktionen ist es wichtig, die Kernkonzepte und -komponenten zu verstehen, aus denen das Lambda-Programmiermodell besteht. Dieser Abschnitt führt Sie durch die grundlegenden Elemente, die Sie kennen müssen, um mit der Entwicklung serverloser Anwendungen mit Lambda zu beginnen.

- Lambda-Funktionen und -Funktionshandler— Eine Lambda-Funktion ist ein kleiner Codeblock, der als Reaktion auf Ereignisse ausgeführt wird. Funktionen sind die grundlegenden Bausteine, die Sie zum Erstellen von Anwendungen verwenden. Funktionshandler sind der Einstiegspunkt für Ereignisobjekte, die Ihr Lambda-Funktionscode verarbeitet.
- Lambda-Ausführungsumgebung und -Laufzeiten: Lambda-Ausführungsumgebungen verwalten die Ressourcen, die zum Ausführen Ihrer Funktion erforderlich sind. Laufzeiten sind die sprachspezifischen Umgebungen, in denen Ihre Funktionen ausgeführt werden.
- Ereignisse und Auslöser- Andere AWS-Services können Ihre Funktionen als Reaktion auf bestimmte Ereignisse aufrufen.
- Lambda-Berechtigungen und -Rollen- Kontrollieren Sie, wer auf Ihre Funktionen zugreifen kann und mit welchen anderen Funktionen AWS-Services Ihre Funktionen interagieren können.

> **Tip**
>
> Wenn Sie die Entwicklung von Serverless-Komponenten zuerst besser verstehen möchten, lesen Sie die Informationen zum Unterschied zwischen herkömmlicher Entwicklung und Serverless-Entwicklung im AWS -Serverless-Entwicklerhandbuch.

Lambda-Funktionen und -Funktionshandler

In Lambda sind Funktionen die grundlegenden Bausteine, mit denen Sie Anwendungen erstellen. Eine Lambda-Funktion ist ein Codeabschnitt, der als Reaktion auf Ereignisse ausgeführt wird, z. B. wenn ein Benutzer eine Schaltfläche auf einer Website auswählt oder eine Datei in einen Amazon Simple Storage Service (Amazon S3) -Bucket hochgeladen wird. Mit langlebigen Funktionen kann Ihr Code die Ausführung zwischen den einzelnen Schritten unterbrechen, sodass der Status automatisch beibehalten wird. Somit eignen sie sich ideal für lang andauernde Workflows

wie die Auftragsabwicklung oder Inhaltsmoderation. Sie können sich eine Funktion als eine Art eigenständiges Programm mit den folgenden Eigenschaften vorstellen.

Ein Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn eine Funktion als Reaktion auf ein Ereignis ausgeführt wird, führt Lambda den Funktionshandler aus. Daten zum Ereignis, das die Ausführung der Funktion verursacht hat, werden direkt an den Handler übergeben. Während der Code in einer Lambda-Funktion mehrere Methoden oder Funktionen enthalten kann, können Lambda-Funktionen nur einen Handler enthalten.

Um eine Lambda-Funktion zu erstellen, bündeln Sie den Funktionscode und die zugehörigen Abhängigkeiten in einem Bereitstellungspaket. Lambda unterstützt zwei Arten von Bereitstellungspaketen: ZIP-Dateiarchive und Container-Images.

- Eine Funktion hat eine bestimmte Aufgabe oder einen bestimmten Zweck.
- Sie wird nur ausgeführt, wenn sie als Reaktion auf bestimmte Ereignisse benötigt wird.
- Sie wird automatisch beendet, wenn sie fertig ist.

Lambda-Ausführungsumgebung und -Laufzeiten

Lambda-Funktionen werden in einer sicheren, isolierten Ausführungsumgebung ausgeführt, die Lambda für Sie verwaltet. Für langlebige Funktionen umfasst die Ausführungsumgebung zusätzliche Komponenten für die Statusverwaltung und die Workflow-Koordination. Die Ausführungsumgebung verwaltet die Prozesse und Ressourcen, die für die Ausführung Ihrer Funktion benötigt werden. Wenn eine Funktion zum ersten Mal aufgerufen wird, erstellt Lambda eine neue Ausführungsumgebung, in der die Funktion ausgeführt werden kann. Nachdem die Ausführung der Funktion abgeschlossen wurde, beendet Lambda die Ausführungsumgebung nicht sofort. Wenn die Funktion erneut aufgerufen wird, kann Lambda die vorhandene Ausführungsumgebung wiederverwenden.

Die Lambda-Ausführungsumgebung enthält außerdem eine Laufzeit, eine sprachspezifische Umgebung, die Ereignisinformationen und Antworten zwischen Lambda und Ihrer Funktion weiterleitet. Lambda bietet eine Reihe von verwalteten Laufzeiten für die gängigsten Programmiersprachen, Sie können aber auch eigene erstellen.

Bei verwalteten Laufzeiten wendet Lambda automatisch Sicherheitsupdates und Patches auf Funktionen an, die die Laufzeit verwenden.

Ereignisse und Auslöser

Sie können eine Lambda-Funktion auch direkt über die Lambda-Konsole, die AWS CLI oder eins der AWS Software Development Kits (SDKs) aufrufen. In einer Produktionsanwendung ist es üblicher, dass Ihre Funktion als Reaktion auf ein bestimmtes Ereignis von einer anderen AWS-Service aufgerufen wird. Sie möchten beispielsweise, dass eine Funktion immer dann ausgeführt wird, wenn ein Element zu einer Amazon-DynamoDB-Tabelle hinzugefügt wird.

Damit Ihre Funktion auf Ereignisse reagiert, richten Sie einen Auslöser ein. Ein Auslöser verbindet Ihre Funktion mit einer Ereignisquelle und Ihre Funktion kann mehrere Auslöser haben. Wenn ein Ereignis eintritt, empfängt Lambda Ereignisdaten als JSON-Dokument und konvertiert es in ein Objekt, das Ihr Code verarbeiten kann. Sie können das folgende JSON-Format für Ihr Ereignis definieren und die Lambda-Laufzeit konvertiert dieses JSON-Dokument in ein Objekt, bevor es an den Handler Ihrer Funktion übergeben wird.

Example Benutzerdefiniertes Lambda-Ereignis

```
{
  "Location": "SEA",
  "WeatherData":{
    "TemperaturesF":{
      "MinTempF": 22,
      "MaxTempF": 78
    },
    "PressuresHPa":{
      "MinPressureHPa": 1015,
      "MaxPressureHPa": 1027
    }
  }
}
```

Stream- und Warteschlangendienste wie Amazon Kinesis oder Amazon SQS verwenden eine Zuordnung der Ereignisquelle anstelle eines Standard-Triggers. Bei Zuordnungen von Ereignisquellen wird die Quelle nach neuen Daten abgefragt, Datensätze gebündelt und anschließend Ihre Funktion mit den gebündelten Ereignissen aufgerufen. Weitere Informationen finden Sie unter Wie unterscheiden sich Zuordnungen von Ereignisquellen von direkten Auslösern.

Um zu verstehen, wie ein Auslöser funktioniert, beginnen Sie mit dem Tutorial Verwenden eines Amazon-S3-Auslösers. Einen allgemeinen Überblick über die Verwendung von Auslösern und

Anweisungen zum Erstellen eines Auslösers mithilfe der Lambda-Konsole finden Sie unter Integration anderer Services.

Lambda-Berechtigungen und -Rollen

Bei Lambda gibt es zwei Haupttypen von Berechtigungen, die Sie konfigurieren müssen:

- Berechtigungen, die Ihre Funktion für den Zugriff auf andere benötigt AWS-Services
- Berechtigungen, die andere Benutzer AWS-Services benötigen, um auf deine Funktion zuzugreifen

In den folgenden Abschnitten werden diese beiden Berechtigungstypen beschrieben und bewährte Methoden für die Anwendung von geringsten Berechtigungen erörtert.

Berechtigungen für Funktionen zum Zugriff auf andere AWS Ressourcen

Lambda-Funktionen müssen häufig auf andere AWS Ressourcen zugreifen und mit ihnen Aktionen ausführen. Eine Funktion könnte beispielsweise Elemente aus einer DynamoDB-Tabelle lesen, ein Objekt in einem S3-Bucket speichern oder in eine Amazon-SQS-Warteschlange schreiben. Um Funktionen die Berechtigungen zu geben, die sie für die Ausführung dieser Aktionen benötigen, verwenden Sie eine Ausführungsrolle.

Eine Lambda-Ausführungsrolle ist eine spezielle Art von AWS Identity and Access Management (IAM-) Rolle, eine Identität, die Sie in Ihrem Konto erstellen und der bestimmte, in einer Richtlinie definierte Berechtigungen zugeordnet sind.

Jede Lambda-Funktion muss eine Ausführungsrolle haben und eine einzelne Rolle kann von mehr als einer Funktion verwendet werden. Wenn eine Funktion aufgerufen wird, übernimmt Lambda die Ausführungsrolle der Funktion und erhält die Berechtigung, die in der Richtlinie der Rolle definierten Aktionen auszuführen.

Wenn Sie eine Funktion in der Lambda-Konsole erstellen, erstellt Lambda automatisch eine Ausführungsrolle für Ihre Funktion. Die Richtlinie der Rolle erteilt Ihrer Funktion grundlegende Berechtigungen zum Schreiben von Protokollausgaben in Amazon CloudWatch Logs. Um Ihrer Funktion die Erlaubnis zu geben, Aktionen mit anderen AWS Ressourcen auszuführen, müssen Sie die Rolle bearbeiten, um die zusätzlichen Berechtigungen hinzuzufügen.

Der einfachste Weg, Berechtigungen hinzuzufügen, besteht darin, eine AWS verwaltete Richtlinie zu verwenden. Verwaltete Richtlinien werden von erstellt und verwaltet AWS und bieten Berechtigungen für viele gängige Anwendungsfälle. Wenn Ihre Funktion beispielsweise CRUD-Operationen an einer

DynamoDB-Tabelle ausführt, können Sie die AmazonDynamoDBFullAccess Richtlinie Ihrer Rolle hinzufügen.

Berechtigungen für andere Benutzer und Ressourcen für den Zugriff auf Ihre Funktion

Um anderen AWS-Service Berechtigungen für den Zugriff auf Ihre Lambda-Funktion zu gewähren, verwenden Sie eine ressourcenbasierte Richtlinie. ??? In IAM werden ressourcenbasierte Richtlinien einer Ressource zugeordnet (in diesem Fall Ihrer Lambda-Funktion). Sie definieren, wer auf die Ressource zugreifen kann und welche Aktionen sie ausführen können.

Damit ein anderer AWS-Service Benutzer Ihre Funktion über einen Trigger aufrufen kann, muss die ressourcenbasierte Richtlinie Ihrer Funktion diesem Dienst die Berechtigung zur Verwendung der Aktion gewähren. `lambda:InvokeFunction` Wenn Sie den Auslöser mit der Konsole erstellen, fügt Lambda diese Berechtigung automatisch für Sie hinzu.

Um anderen AWS Benutzern die Erlaubnis zu erteilen, auf Ihre Funktion zuzugreifen, können Sie dies in der ressourcenbasierten Richtlinie Ihrer Funktion genauso definieren wie für eine andere oder eine Ressource. AWS-Service Sie können auch eine identitätsbasierte Richtlinie verwenden, die dem Benutzer zugeordnet ist.

Bewährte Methoden für Lambda-Berechtigungen

Wenn Sie Berechtigungen mithilfe von IAM-Richtlinien festlegen, empfiehlt es sich aus Sicherheitsgründen nur die zum Ausführen einer Aufgabe erforderlichen Berechtigungen zu erteilen. Dies wird als Prinzip der geringsten Berechtigungen bezeichnet. Um mit der Erteilung von Berechtigungen für Ihre Funktion zu beginnen, können Sie sich für eine AWS verwaltete Richtlinie entscheiden. Verwaltete Richtlinien können die schnellste und einfachste Methode sein, um Berechtigungen für die Ausführung einer Aufgabe zu erteilen. Sie können jedoch auch andere Berechtigungen beinhalten, die Sie nicht benötigen. Wenn Sie von der frühen Entwicklungsphase über die Testphase bis hin zur Produktion voranschreiten, empfehlen wir Ihnen, die Berechtigungen auf die erforderlichen zu reduzieren, indem Sie Ihre eigenen vom Kunden verwalteten Richtlinien definieren.

Das gleiche Prinzip gilt für die Erteilung von Berechtigungen für den Zugriff auf Ihre Funktion mithilfe einer ressourcenbasierten Richtlinie. Wenn Sie beispielsweise Amazon S3 die Erlaubnis geben möchten, Ihre Funktion aufzurufen, empfiehlt es sich, den Zugriff auf einzelne Buckets oder insbesondere auf Buckets zu beschränken AWS-Konten, anstatt dem S3-Service pauschale Berechtigungen zu erteilen.

Ausführen von Code mit Lambda

Wenn Sie eine Lambda-Funktion schreiben, erstellen Sie Code, der in einer einzigartigen serverlosen Umgebung ausgeführt wird. Zu verstehen, wie Lambda Ihren Code tatsächlich ausführt, beinhaltet zwei wichtige Aspekte: das Programmiermodell, das definiert, wie Ihr Code mit Lambda interagiert, und den Lebenszyklus der Ausführungsumgebung, der bestimmt, wie Lambda die Laufzeitumgebung Ihres Codes verwaltet.

Das Lambda-Programmiermodell

Das Programmiermodell dient als allgemeines Regelwerk dafür, wie Lambda mit Ihrem Code arbeitet, unabhängig davon, ob Sie in Python, Java oder einer anderen unterstützten Sprache schreiben. Das Programmiermodell umfasst Ihre Laufzeit und Ihren Handler.

Für Funktionen ohne dauerhafte Ausführung:

1. Lambda empfängt ein Ereignis.
2. Lambda verwendet die Laufzeit, um das Ereignis in einem Format vorzubereiten, das Ihr Code verwenden kann.
3. Die Laufzeit sendet das formatierte Ereignis an Ihren Handler.
4. Ihr Handler verarbeitet das Ereignis mit dem Code, den Sie geschrieben haben.

Wesentlich für dieses Modell ist der Handler, an den Lambda Ereignisse sendet, die von Ihrem Code verarbeitet werden sollen. Betrachten Sie ihn als Einstiegspunkt für Ihren Code. Wenn Lambda ein Ereignis empfängt, leitet es dieses Ereignis und einige Kontextinformationen an Ihren Handler weiter. Der Handler führt dann Ihren Code aus, um diese Ereignisse zu verarbeiten – beispielsweise kann er eine Datei lesen, wenn sie in Amazon S3 hochgeladen wird, ein Bild analysieren oder eine Datenbank aktualisieren. Sobald Ihr Code die Verarbeitung eines Ereignisses abgeschlossen hat, ist der Handler bereit, das nächste Ereignis zu verarbeiten.

Das Lambda-Ausführungsmodell

Während das Programmiermodell definiert, wie Lambda mit Ihrem Code interagiert, ist die Ausführungsumgebung der Ort, an dem Lambda Ihre Funktion tatsächlich ausführt — es handelt sich um einen sicheren, isolierten Rechenraum, der speziell für Ihre Funktion erstellt wurde.

Jede Umgebung folgt einem Lebenszyklus:

1. Initialisierung: Einrichtung der Umgebung und Laden des Codes
2. Aufruf: Einmalige Ausführung des Funktionscodes
3. Herunterfahren: Bereinigung der Umgebung

Diese Umgebung übernimmt wichtige Aspekte der Ausführung Ihrer Funktion. Sie stellt Ihrer Funktion Speicher und ein `/tmp`-Verzeichnis für die temporäre Speicherung zur Verfügung.

Grundlegendes zum Lambda-Programmiermodell

Lambda-Ereignisfunktionen werden standardmäßig bis zu 15 Minuten pro Aufruf ausgeführt. Asynchrone Aufrufe und ESM-Aufrufe (Event Source Mapping) auf Lambda Managed Instances können bis zu 90 Minuten lang ausgeführt werden. Durable Functions können bis zu einem Jahr lang ausgeführt werden. Durable Functions haben zwar gemeinsame Kernkonzepte, bieten aber zusätzliche Funktionen für langanhaltende, statusbehaftete Workflows.

Lambda stellt ein Programmiermodell bereit, das allen Laufzeitumgebungen gemeinsam ist. Das Programmiermodell definiert die Schnittstelle zwischen Ihrem Code und dem Lambda-System. Sie teilen Lambda den Einstiegspunkt für Ihre Funktion mit, indem Sie einen Handler in der Funktionskonfiguration definieren. Die Laufzeit übergibt Objekte an den Handler, die das Aufrufereignis und den Kontext, wie z. B. den Funktionsnamen und die Anforderungs-ID, enthalten.

Bei Durable Functions erhält der Handler außerdem ein DurableContext Objekt, das Folgendes bereitstellt:

- Überprüfen Sie die Funktionen mithilfe von step ()
- Warten Sie, bis der Status mit wait () und wait ForCallback () verwaltet wird
- Automatische Statuspersistenz zwischen Aufrufen

Wenn der Handler die Verarbeitung des ersten Ereignisses beendet hat, sendet die Laufzeit ihm ein anderes. Bei dauerhaften Funktionen kann der Handler die Ausführung zwischen den Schritten unterbrechen, und Lambda speichert automatisch den Status und stellt ihn wieder her, wenn die Funktion wieder aufgenommen wird. Die Klasse der Funktion bleibt im Speicher, sodass Clients und Variablen, die außerhalb der Handler-Methode in Initialisierungscodedeklariert sind, wiederverwendet werden können. Um die Verarbeitungszeit bei nachfolgenden Ereignissen zu verkürzen, erstellen Sie während der Initialisierung wiederverwendbare Ressourcen wie AWS SDK-Clients. Nach der Initialisierung kann jede Instance Ihrer Funktion Tausende von Anfragen verarbeiten.

Ihre Funktion hat auch Zugriff auf den lokalen Speicher im `/tmp`-Verzeichnis, einen temporären Zwischenspeicher, der für mehrere Aufrufe verwendet werden kann. Weitere Informationen finden Sie unter Ausführungsumgebung.

Wenn die AWS X-Ray -Ablaufverfolgung aktiviert ist, zeichnet die Laufzeitumgebung separate Untersegmente für die Initialisierung und Ausführung auf.

Die Runtime erfasst die Protokollausgabe Ihrer Funktion und sendet sie an Amazon CloudWatch Logs. Neben der Protokollierung der Ausgabe Ihrer Funktion protokolliert die Laufzeitumgebung auch Einträge, wenn der Funktionsaufruf gestartet und beendet wird. Dazu gehört ein Berichtsprotokoll mit der Anforderungskennung, der fakturierten Dauer, der Initialisierungsdauer und weiteren Details. Wenn Ihre Funktion einen Fehler ausgibt, gibt die Laufzeit diesen Fehler an den Aufrufer zurück.

> **Note**
>
> Die Protokollierung unterliegt den CloudWatch Log-Kontingenten. Protokolldaten können durch Ablehnung verloren gehen, oder, in einigen Fällen, wenn eine Instance Ihrer Funktion gestoppt wird.

Hauptunterschiede bei Durable Functions:

- Der Status wird zwischen den Schritten automatisch beibehalten
- Funktionen können die Ausführung unterbrechen, ohne Ressourcen zu verbrauchen
- Bei einem Fehler werden die Schritte automatisch wiederholt
- Der Fortschritt wird anhand von Checkpoints verfolgt

Lambda skaliert Ihre Funktion, indem zusätzliche Instances davon ausgeführt werden, wenn der Bedarf steigt und indem Instances beendet werden, wenn der Bedarf sinkt. Dieses Modell führt zu Variationen in der Anwendungsarchitektur, wie zum Beispiel:

- Sofern nicht anders angegeben, werden eingehende Anforderungen nicht in der richtigen Reihenfolge oder gleichzeitig verarbeitet.
- Verlassen Sie sich nicht darauf, dass die Instances Ihrer Funktion langlebig sind, sondern speichern Sie den Zustand Ihrer Anwendung an anderer Stelle.

- Verwenden Sie lokale Speicher und Objekte auf Klassenebene, um die Leistung zu steigern, aber halten Sie die Größe Ihres Bereitstellungspakets und die Datenmenge, die Sie in die Ausführungsumgebung übertragen, möglichst gering.

In den folgenden Kapiteln finden Sie eine praktische Einführung in das Programmiermodell in Ihrer bevorzugten Programmiersprache.

- Erstellen von Lambda-Eventfunktionen mit Node.js
- Lambda-Ereignisfunktionen mit Python erstellen
- Lambda-Ereignisfunktionen mit Ruby erstellen
- Lambda-Ereignisfunktionen mit Java erstellen
- Erstellen von Lambda-Eventfunktionen mit Go
- Lambda-Ereignisfunktionen mit C# erstellen
- Lambda-Ereignisfunktionen erstellen mit PowerShell

Verständnis des Lebenszyklus der Lambda-Ausführungsumgebung

In Lambda-Ausführungsumgebungen werden Funktionsaufrufe standardmäßig bis zu 15 Minuten lang ausgeführt, und langlebige Funktionen können bis zu einem Jahr lang ausgeführt werden. Dauerhafte Funktionen haben denselben grundlegenden Lebenszyklus und bieten zusätzliche Funktionen zur Statusverwaltung für Workflows mit langer Laufzeit.

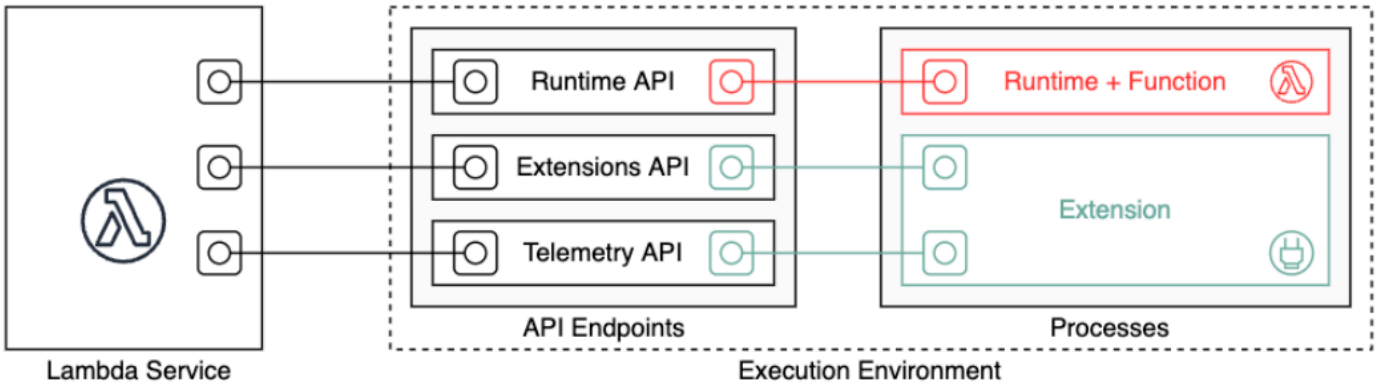
Lambda ruft Ihre Funktion in einer Ausführungsumgebung auf, die eine sichere und isolierte Laufzeitumgebung bereitstellt. Die Ausführungsumgebung verwaltet die Ressourcen, die zum Ausführen Ihrer Funktion erforderlich sind. Die Ausführungsumgebung bietet auch Lebenszyklusunterstützung für die Laufzeit der Funktion und alle externen Erweiterungen die mit Ihrer Funktion verknüpft sind.

Für langlebige Funktionen umfasst die Ausführungsumgebung zusätzliche Komponenten für:

- Persistenz des Zustands zwischen den Schritten
- Verwaltung von Kontrollpunkten
- Warten Sie die staatliche Koordination ab
- Verfolgung der Fortschritte

> **Ausführungsumgebung für Lambda Managed Instances**
>
> Wenn Sie Lambda Managed Instances verwenden, weist die Ausführungsumgebung wichtige Unterschiede zu Lambda-Funktionen (Standard) auf. Verwaltete Instances unterstützen gleichzeitige Aufrufe, verwenden ein anderes Lebenszyklusmodell und werden auf einer kundeneigenen Infrastruktur ausgeführt. Ausführliche Informationen zur Ausführungsumgebung von Managed Instances finden Sie unter. Grundlegendes zur Lambda Managed Instance-Ausführungsumgebung

Die Laufzeitumgebung der Funktion kommuniziert Lambda mit der Laufzeit-API. Erweiterungen kommunizieren mit Lambda über die Erweiterungs-API. Erweiterungen können mithilfe der Telemetrie-API auch Protokollnachrichten und andere Telemetriedaten von der Funktion empfangen.



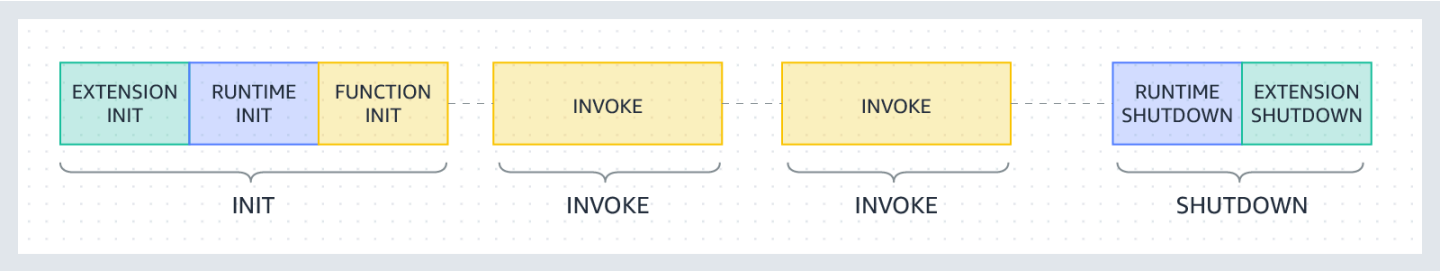
Wenn Sie Ihre Lambda-Funktion erstellen, geben Sie Konfigurationsinformationen an, z. B. den verfügbaren Arbeitsspeicher und die maximal zulässige Ausführungszeit für Ihre Funktion. Lambda verwendet diese Informationen, um die Ausführungsumgebung einzurichten.

Die Laufzeit der Funktion und jede externe Erweiterung sind Prozesse, die innerhalb der Ausführungsumgebung ausgeführt werden. Berechtigungen, Ressourcen, Anmeldeinformationen und Umgebungsvariablen werden von der Funktion und den Erweiterungen gemeinsam genutzt.

Themen

- Lebenszyklus der Lambda-Ausführungsumgebung
- Kaltstarts und Latenzzeiten
- Reduzierung von Kaltstarts mit bereitgestellter Parallelität
- Optimierung der statischen Initialisierung

Lebenszyklus der Lambda-Ausführungsumgebung



Jede Phase beginnt mit einem Ereignis, das Lambda an die Laufzeit und an alle registrierten Erweiterungen sendet. Die Laufzeit und jede Erweiterung zeigen den Abschluss durch Senden einer `Next`-API-Anfrage an. Lambda friert die Ausführungsumgebung ein, wenn die Laufzeit und jede Erweiterung abgeschlossen sind und keine ausstehenden Ereignisse vorhanden sind.

Die Lebenszyklusphasen für Durable Functions umfassen:

- Init: Standardinitialisierung plus Einrichtung eines dauerhaften Zustands
- Invoke: Kann die Ausführung mehrerer Schritte mit automatischem Checkpointing beinhalten
- Warte: Die Funktion kann die Ausführung unterbrechen, ohne Ressourcen zu verbrauchen
- Fortfahren: Die Funktion wird vom letzten Checkpoint aus neu gestartet
- Herunterfahren: Wiederherstellung des dauerhaften Zustands und der Ressourcen

Topics

- Init-Phase
- Fehler in der Initialisierungsphase
- Wiederherstellungsphase (SnapStart nur Lambda)
- Invoke-Phase
- Fehler während der Aufrufphase
- Shutdown-Phase

Init-Phase

Lambda führt in der `Init`-Phase drei Aufgaben aus:

- Alle Erweiterungen starten (`Extension init`)
- Laufzeit-Bootstrap (`Runtime init`)

- Ausführen des statischen Codes der Funktion (`Function init`)
- Alle Runtime-Hooks vor dem Checkpoint ausführen (nur Lambda) SnapStart

Die `Init`-Phase endet, wenn die Laufzeit und alle Erweiterungen signalisieren, dass sie bereit sind, indem sie eine `Next`-API-Anforderung senden. Die `Init`-Phase ist auf 10 Sekunden begrenzt. Wenn alle drei Aufgaben nicht innerhalb von 10 Sekunden abgeschlossen werden, versucht Lambda die `Init`-Phase zum Zeitpunkt des ersten Funktionsaufrufs mit dem konfigurierten Funktionstimeout erneut.

Wenn SnapStart aktiviert ist, findet die `Init`-Phase statt, wenn Sie eine Funktionsversion veröffentlichen. Lambda speichert einen Snapshot des Arbeitsspeichers und des Festplattenzustands der initialisierten Ausführungsumgebung, speichert den verschlüsselten Snapshot und speichert ihn im Cache für den Zugriff mit geringer Latenz. Wenn Sie einen Laufzeit-Hook vor dem Prüfpunkt haben, wird der Code am Ende der `Init`-Phase ausgeführt.

> **Note**
>
> Das 10-Sekunden-Timeout gilt nicht für Funktionen, die bereitgestellte Parallelität oder Lambda Managed SnapStart Instances verwenden. Für die Funktionen Provisioned Concurrency und Managed Instances kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. SnapStart Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist.

Wenn Sie die bereitgestellte Gleichzeitigkeit verwenden, initialisiert Lambda die Ausführungsumgebung, wenn Sie die PC-Einstellungen für eine Funktion konfigurieren. Lambda stellt außerdem sicher, dass initialisierte Ausführungsumgebungen vor Aufrufen immer verfügbar sind. Möglicherweise treten Lücken zwischen der Aufruf- und der Initialisierungsphase Ihrer Funktion auf. Abhängig von der Laufzeit- und Speicherkonfiguration Ihrer Funktion kann es auch zu einer variablen Latenz beim ersten Aufruf in einer initialisierten Ausführungsumgebung kommen.

Bei Funktionen, die On-Demand-Parallelität verwenden, initialisiert Lambda gelegentlich Ausführungsumgebungen vor Aufrufanforderungen. In diesem Fall stellen Sie möglicherweise auch eine Zeitlücke zwischen der Initialisierung und der Aufrufphase Ihrer Funktion fest. Wir empfehlen Ihnen, sich nicht von diesem Verhalten abhängig zu machen.

Fehler in der Initialisierungsphase

Wenn eine Funktion in der `Init`-Phase abstürzt oder ein Timeout auftritt, gibt Lambda Fehlerinformationen im `INIT_REPORT`-Protokoll aus.

Example– INIT_REPORT-Protokoll für Timeout

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example– INIT_REPORT-Protokoll für den Erweiterungsfehler

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

Wenn die `Init` Phase erfolgreich ist, gibt Lambda das `INIT_REPORT` Protokoll nicht aus, es sei denn SnapStart, die bereitgestellte Parallelität ist aktiviert. SnapStart und bereitgestellte Parallelitätsfunktionen werden immer ausgegeben. `INIT_REPORT` Weitere Informationen finden Sie unter Überwachung für Lambda SnapStart.

Wiederherstellungsphase (SnapStart nur Lambda)

Wenn Sie eine SnapStartFunktion zum ersten Mal aufrufen und die Funktion skaliert wird, nimmt Lambda neue Ausführungsumgebungen aus dem persistenten Snapshot wieder auf, anstatt die Funktion von Grund auf neu zu initialisieren. Wenn Sie über einen Laufzeit-Hook nach der Wiederherstellung verfügen, wird der Code am Ende der `Restore`-Phase ausgeführt. Die Dauer von Laufzeit-Hooks nach der Wiederherstellung wird Ihnen in Rechnung gestellt. Die Laufzeit muss geladen werden und Laufzeit-Hooks nach der Wiederherstellung müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine. SnapStartTimeoutException Wenn die `Restore`-Phase abgeschlossen ist, ruft Lambda den Funktionshandler (Invoke-Phase) auf.

Fehler in der Wiederherstellphase

Wenn die `Restore`-Phase fehlschlägt, gibt Lambda Fehlerinformationen im `RESTORE_REPORT`-Protokoll aus.

Example– RESTORE_REPORT-Protokoll für Timeout

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example– RESTORE_REPORT-Protokoll für einen Laufzeit-Hook-Fehler

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

Weitere Informationen zum `RESTORE_REPORT`-Protokoll finden Sie unter Überwachung für Lambda SnapStart.

Invoke-Phase

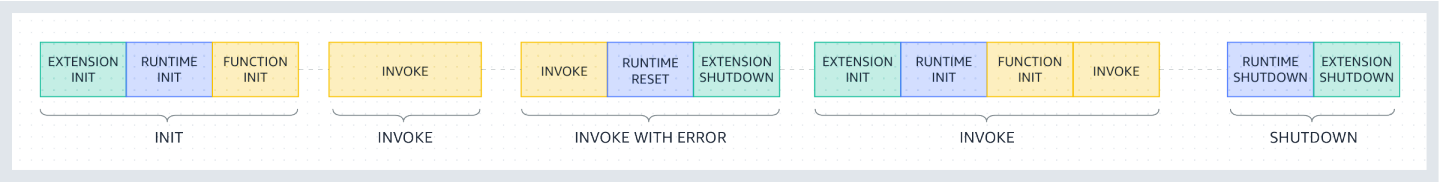
Wenn eine Lambda-Funktion als Antwort auf eine `Next`-API-Anforderung aufgerufen wird, sendet Lambda ein `Invoke`-Ereignis an die Laufzeit und an jede Erweiterung.

Die Timeout-Einstellung der Funktion begrenzt die Dauer der gesamten `Invoke`-Phase. Wenn Sie beispielsweise die Zeitüberschreitung für die Funktion auf 360 Sekunden festlegen, müssen die Funktion und alle Erweiterungen innerhalb von 360 Sekunden abgeschlossen werden. Beachten Sie, dass es keine unabhängige Post-Invoke-Phase gibt. Die Dauer ist die Summe der gesamten Aufrufzeit (Laufzeit + Erweiterungen) und wird erst berechnet, wenn die Funktion und alle Erweiterungen vollständig ausgeführt wurden.

Die Invoke-Phase endet nach der Laufzeit und alle Erweiterungen signalisieren durch Senden einer `Next`-API-Anforderung dass sie abgeschlossen sind.

Fehler während der Aufrufphase

Wenn die Lambda-Funktion während der `Invoke`-Phase abstürzt oder das Zeitlimit überschreitet, setzt Lambda die Ausführungsumgebung zurück. Das folgende Diagramm veranschaulicht das Verhalten der Lambda-Ausführungsumgebung bei einem Aufruffehler:



In folgendem Diagramm:

- Ist die erste Phase die INIT-Phase, die ohne Fehler läuft.
- Ist die zweite Phase die INVOKE-Phase, die ohne Fehler läuft.
- Angenommen, Ihre Funktion stößt irgendwann auf einen Aufruffehler (häufige Ursachen sind Funktions-Timeouts, Laufzeitfehler, Speichererschöpfung, VPC-Konnektivitätsprobleme, Berechtigungsfehler, Parallelitätsbeschränkungen und verschiedene Konfigurationsprobleme).

Eine vollständige Liste möglicher Aufruffehler finden Sie unter the section called “Aufruf”. Die dritte Phase mit der Bezeichnung INVOKE MIT FEHLER veranschaulicht dieses Szenario.

Wenn dies geschieht, führt der Lambda-Service einen Neustart durch. Der Reset verhält sich wie ein `Shutdown`-Ereignis. Zuerst beendet Lambda die Laufzeit und sendet dann ein `Shutdown`-Ereignis an jede registrierte externe Erweiterung. Das Ereignis enthält den Grund für das Abschalten. Wenn diese Umgebung für einen neuen Aufruf verwendet wird, initialisiert Lambda die Erweiterung und die Laufzeit als Teil des nächsten Aufrufers erneut.

Beachten Sie, dass der Lambda-Reset den `/tmp` Verzeichnisinhalt nicht vor der nächsten Init-Phase löscht. Dieses Verhalten stimmt mit der regulären Shutdown-Phase überein.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können Sie geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollnachrichten und Trace-Segmenten feststellen, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Wenn die Systemprotokollkonfiguration Ihrer Funktion auf Klartext eingestellt ist, wirkt sich diese Änderung auf die Protokollnachrichten aus, die in CloudWatch Logs erfasst werden, wenn bei Ihrer Funktion ein Aufruffehler auftritt. Die folgenden Beispiele zeigen Protokollausgaben sowohl in alten als auch in neuen Formaten.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Example CloudWatch Protokolliert die Protokollausgabe (Laufzeit- oder Erweiterungsabsturz) — alter Stil

```
START RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Version: $LATEST
RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Error: Runtime exited without
 providing a reason
Runtime.ExitError
END RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1
REPORT RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Duration: 933.59 ms Billed
 Duration: 934 ms Memory Size: 128 MB Max Memory Used: 9 MB
```

Example CloudWatch Protokolliert die Protokollausgabe (Funktions-Timeout) im alten Stil

```
START RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Version: $LATEST
2024-03-04T17:22:38.033Z b70435cc-261c-4438-b9b6-efe4c8f04b21 Task timed out after
 3.00 seconds
END RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21
REPORT RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Duration: 3004.92 ms Billed
 Duration: 3117 ms Memory Size: 128 MB Max Memory Used: 33 MB Init Duration: 111.23
 ms
```

Das neue Format für CloudWatch Protokolle enthält ein zusätzliches `status` Feld in der `REPORT` Zeile. Im Falle eines Laufzeit- oder Erweiterungsabsturzes enthält die `REPORT`-Zeile auch ein Feld `ErrorType`.

Example CloudWatch Protokolliert die Protokollausgabe (Laufzeit- oder Erweiterungsabsturz) — neuer Stil

```
START RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Version: $LATEST
END RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd
REPORT RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Duration: 133.61 ms Billed
 Duration: 214 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 80.00
 ms Status: error Error Type: Runtime.ExitError
```

Example CloudWatch Protokolliert die Protokollausgabe (Funktions-Timeout) — neuer Stil

```
START RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Version: $LATEST
END RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda
REPORT RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Duration: 3016.78 ms Billed
 Duration: 3101 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 84.00
 ms Status: timeout
```

- Die vierte Phase stellt die INVOKE-Phase unmittelbar nach einem Aufruffehler dar. Hier initialisiert Lambda die Umgebung erneut, indem es die INIT-Phase erneut ausführt. Dies wird als unterdrückte Initialisierung bezeichnet.

  Wenn unterdrückte Inits auftreten, meldet Lambda nicht explizit eine zusätzliche INIT-Phase in Logs. CloudWatch Stattdessen stellen Sie möglicherweise fest, dass die Dauer in der REPORT-Zeile eine zusätzliche INIT-Dauer plus die INVOKE-Dauer umfasst. Nehmen wir zum Beispiel an, Sie sehen die folgenden Logins: CloudWatch

```
2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
2022-12-20T01:00:02.500-08:00 END RequestId: XXX
2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
```

In diesem Beispiel beträgt der Unterschied zwischen den Zeitstempeln BERICHT und START 2,5 Sekunden. Dies entspricht nicht der angegebenen Dauer von 3 022,91 Millsekunden, da es die zusätzliche INIT (unterdrückte Initialisierung), die Lambda ausgeführt hat, nicht berücksichtigt. In diesem Beispiel können Sie ableiten, dass die tatsächliche INVOKE-Phase 2,5 Sekunden gedauert hat.

Für mehr Einblick in dieses Verhalten können Sie die Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API verwenden. Die Telemetrie-API gibt `INIT_START`, `INIT_RUNTIME_DONE` und `INIT_REPORT`-Ereignisse mit `phase=invoke` immer dann aus, wenn unterdrückte Initialisierungen zwischen den Aufrufphasen auftreten.

- Die fünfte Phase stellt die SHUTDOWN-Phase dar, die fehlerfrei abläuft.

Shutdown-Phase

Wenn Lambda dabei ist, die Laufzeit herunterzufahren, sendet es ein `Shutdown`-Ereignis an jede registrierte externe Erweiterung. Erweiterungen können diese Zeit für abschließende Bereinigungsaufgaben verwenden. Das `Shutdown`-Ereignis ist eine Antwort auf eine `Next`-API-Anforderung.

Duration Limit (Begrenzung der Dauer): Die maximale Dauer der `Shutdown`-Phase hängt von der Konfiguration der registrierten Erweiterungen ab:

- 0 ms – Eine Funktion ohne registrierte Erweiterungen
- 500 ms – Eine Funktion mit einer registrierten internen Erweiterung
- 2.000 ms – Eine Funktion mit einer oder mehreren registrierten externen Erweiterungen

Wenn die Laufzeit oder eine Erweiterung nicht innerhalb des Limits auf das `Shutdown`-Ereignis reagiert, beendet Lambda den Prozess mit einem `SIGKILL`-Signal.

Nachdem die Funktion und alle Erweiterungen abgeschlossen sind, behält Lambda die Ausführungsumgebung für einige Zeit im Vorgriff auf einen anderen Funktionsaufruf bei. Lambda beendet jedoch alle paar Stunden die Ausführungsumgebungen, um Laufzeitaktualisierungen und -

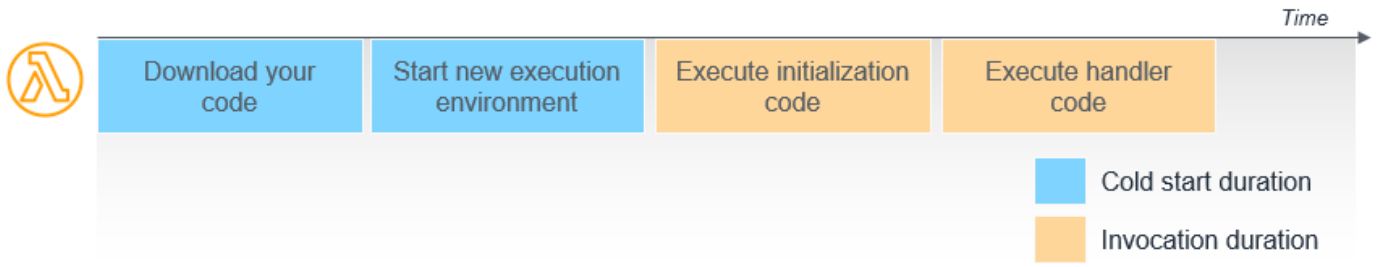
wartungen zu ermöglichen – selbst bei Funktionen, die kontinuierlich aufgerufen werden. Sie sollten nicht davon ausgehen, dass die Ausführungsumgebung auf unbestimmte Zeit bestehen bleibt. Weitere Informationen finden Sie unter Implementieren der Zustandslosigkeit in Funktionen.

Wenn die Funktion erneut aufgerufen wird, taut Lambda die Umgebung zur Wiederverwendung auf. Die Wiederverwendung der Ausführungsumgebung hat folgende Auswirkungen:

- Objekte, die außerhalb der Handler-Methode der Funktion deklariert sind, bleiben initialisiert und bieten eine zusätzliche Optimierung, wenn die Funktion erneut aufgerufen wird. Beispiel: Falls Ihre Lambda-Funktion eine Datenbankverbindung herstellt, statt die Verbindung neu herzustellen, wird die ursprüngliche Verbindung bei nachfolgenden Aufrufen verwendet. Wir empfehlen, dass Sie Ihrem Code eine Logik hinzufügen, um zu prüfen, ob eine Verbindung besteht, bevor Sie eine neue erstellen.
- Jede Ausführungsumgebung stellt Speicherplatz von 512 MB bis 10 240 MB, in 1-MB-Schritten, im Verzeichnis `/tmp` zur Verfügung. Der Inhalt des Verzeichnisses bleibt bestehen, wenn die Ausführungsumgebung eingefroren ist, und bietet einen temporären Zwischenspeicher, der für mehrere Aufrufe verwendet werden kann. Sie können zusätzlichen Code hinzufügen, um zu prüfen, ob der Cache die gespeicherten Daten enthält. Weitere Informationen zu Beschränkungen der Bereitstellungsgröße finden Sie unter Lambda-Kontingente.
- Hintergrundprozesse oder Callbacks, die von Ihrer Lambda-Funktion initiiert wurden und nicht abgeschlossen wurden, wenn die Funktion beendet wurde, werden wieder aufgenommen, wenn Lambda die Ausführungsumgebung wiederverwendet. Sie sollten sicherstellen, dass alle Hintergrundprozesse oder Callbacks in Ihrem Code abgeschlossen sind, bevor der Code beendet wird.

Kaltstarts und Latenzzeiten

Wenn Lambda eine Anfrage zur Ausführung einer Funktion über die Lambda-API erhält, bereitet der Dienst zunächst eine Ausführungsumgebung vor. Während dieser Initialisierungsphase lädt der Dienst Ihren Code herunter, startet die Umgebung und führt jeglichen Initialisierungscode außerhalb des Haupt-Handlers aus. Schließlich führt Lambda den Handler-Code aus.



In diesem Diagramm werden die ersten beiden Schritte des Herunterladens des Codes und der Einrichtung der Umgebung häufig als „Kaltstart“ bezeichnet. Diese Zeit wird Ihnen in Rechnung gestellt und verlängert die Gesamtdauer Ihres Aufrufs.

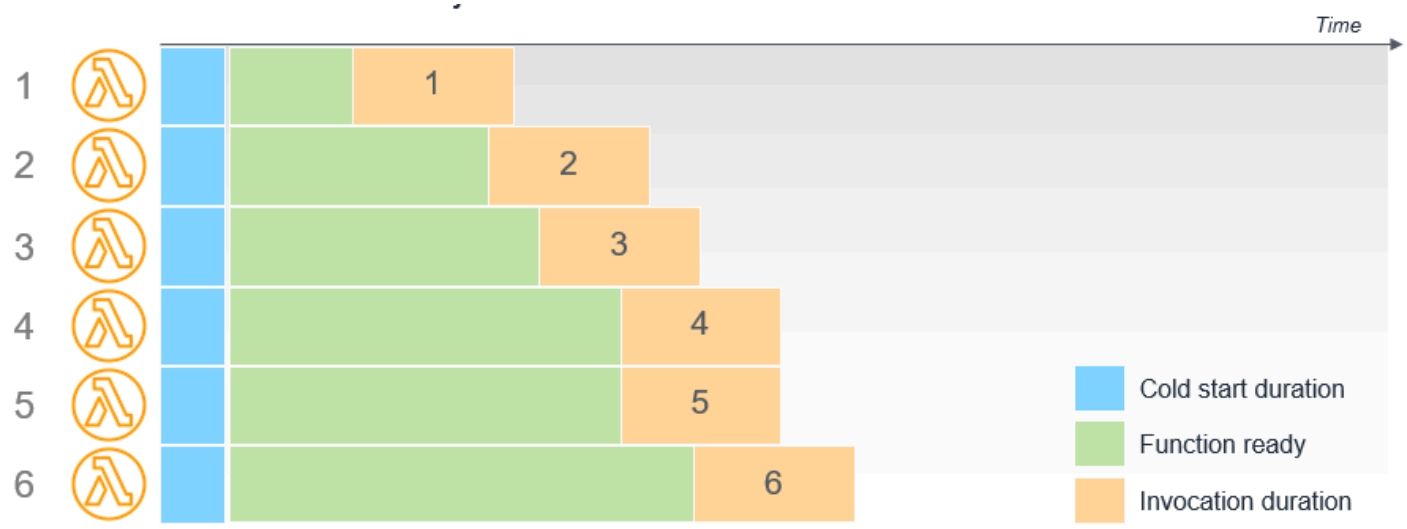
Nachdem der Aufruf abgeschlossen ist, wird die Ausführungsumgebung eingefroren. Um die Ressourcenverwaltung und die Leistung zu verbessern, behält Lambda die Ausführungsumgebung für einen bestimmten Zeitraum bei. Wenn während dieser Zeit eine andere Anfrage für dieselbe Funktion eingeht, kann Lambda die Umgebung wiederverwenden. Diese zweite Anfrage wird in der Regel schneller abgeschlossen, da die Ausführungsumgebung bereits vollständig eingerichtet ist. Dies wird als „Warmstart“ bezeichnet.

Kaltstarts treten in der Regel bei weniger als 1 % der Aufrufe auf. Die Dauer eines Kaltstarts variiert zwischen unter 100 ms und über 1 Sekunde. Im Allgemeinen kommen Kaltstarts in Entwicklungs- und Testfunktionen häufiger vor als in Produktions-Workloads. Das liegt daran, dass Entwicklungs- und Testfunktionen in der Regel seltener aufgerufen werden.

Reduzierung von Kaltstarts mit bereitgestellter Parallelität

Wenn Sie vorhersehbare Startzeiten von Funktionen für Ihren Workload benötigen, ist bereitgestellte Parallelität die empfohlene Lösung, um eine möglichst geringe Latenz sicherzustellen. Mit diesem Feature werden Ausführungsumgebungen vorinitialisiert, was die Anzahl der Kaltstarts reduziert.

Für eine Funktion mit einer bereitgestellten Parallelität von 6 werden beispielsweise 6 Ausführungsumgebungen vorbereitet.



Optimierung der statischen Initialisierung

Die statische Initialisierung erfolgt, bevor der Handlercode in einer Funktion ausgeführt wird. Dies ist der von Ihnen bereitgestellte Initialisierungscode, der sich außerhalb des Haupt-Handlers befindet. Dieser Code wird häufig verwendet, um Bibliotheken und Abhängigkeiten zu importieren, Konfigurationen einzurichten und Verbindungen zu anderen Diensten zu initialisieren.

Das folgende Python-Beispiel zeigt den Import und die Konfiguration von Modulen sowie die Erstellung des Amazon-S3-Clients während der Initialisierungsphase, bevor die `lambda_handler`-Funktion während des Aufrufs ausgeführt wird.

```
import os
import json
import cv2
import logging
import boto3

s3 = boto3.client('s3')
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

  # Handler logic...
```

Der größte Faktor für die Latenz vor der Funktionsausführung ist der Initialisierungscode. Dieser Code wird ausgeführt, wenn eine neue Ausführungsumgebung zum ersten Mal erstellt wird. Der

Initialisierungscode wird nicht erneut ausgeführt, wenn ein Aufruf eine warme Ausführungsumgebung verwendet. Faktoren, die die Latenz des Initialisierungscodes beeinflussen, sind unter anderem:

- Die Größe des Funktionspakets in Bezug auf importierte Bibliotheken und Abhängigkeiten sowie Lambda-Schichten.
- Die Menge an Code und die Initialisierungsarbeit.
- Die Leistung von Bibliotheken und anderen Diensten beim Aufbau von Verbindungen und anderen Ressourcen.

Es gibt eine Reihe von Maßnahmen, mit denen Entwickler die Latenzzeit bei der statischen Initialisierung optimieren können. Wenn eine Funktion viele Objekte und Verbindungen hat, können Sie eine einzelne Funktion möglicherweise in mehrere spezialisierte Funktionen umstrukturieren. Diese sind einzeln kleiner und haben jeweils weniger Initialisierungscode.

Es ist wichtig, dass Funktionen nur die Bibliotheken und Abhängigkeiten importieren, die sie benötigen. Wenn Sie beispielsweise nur Amazon DynamoDB im AWS SDK verwenden, können Sie anstelle des gesamten SDK einen einzelnen Service verlangen. Vergleichen Sie die folgenden drei Beispiele:

```
// Instead of const AWS = require('aws-sdk'), use:
const DynamoDB = require('aws-sdk/clients/dynamodb')

// Instead of const AWSXRay = require('aws-xray-sdk'), use:
const AWSXRay = require('aws-xray-sdk-core')

// Instead of const AWS = AWSXRay.captureAWS(require('aws-sdk')), use:
const dynamodb = new DynamoDB.DocumentClient()
AWSXRay.captureAWSClient(dynamodb.service)
```

Die statische Initialisierung ist auch oft der beste Ort, um Datenbankverbindungen zu öffnen, damit eine Funktion Verbindungen über mehrere Aufrufe derselben Ausführungsumgebung wiederverwenden kann. Möglicherweise verfügen Sie jedoch über eine große Anzahl von Objekten, die nur in bestimmten Ausführungspfaden in Ihrer Funktion verwendet werden. In diesem Fall können Sie Variablen im globalen Bereich langsam laden, um die Dauer der statischen Initialisierung zu reduzieren.

Vermeiden Sie globale Variablen für kontextspezifische Informationen. Wenn Ihre Funktion über eine globale Variable verfügt, die nur für die Lebensdauer eines einzelnen Aufrufs verwendet und für den

nächsten Aufruf zurückgesetzt wird, verwenden Sie einen Variablenbereich, der für den Handler lokal ist. Dadurch wird nicht nur verhindert, dass globale Variablen bei Aufrufen verloren gehen, sondern verbessert auch die Leistung der statischen Initialisierung.

Erstellen ereignisgesteuerter Architekturen mit Lambda

Ein Ereignis ist alles, was das Ausführen einer Lambda-Funktion auslöst. Ereignisse können eine Lambda-Funktion auf zwei Arten auslösen: durch direkten Aufruf (Push) und durch Zuordnungen von Ereignisquellen (Pull).

Viele AWS Dienste können Ihre Lambda-Funktionen direkt aufrufen. Diese Services leiten Ereignisse an Ihre Lambda-Funktion weiter. Ereignisse, die Funktionen auslösen, können fast alles sein, von einer HTTP-Anfrage über API Gateway, einem von EventBridge Scheduler verwalteten Zeitplan, einem AWS IoT Ereignis oder einem Amazon S3 S3-Ereignis. Bei der Zuordnung von Ereignisquellen ruft Lambda aktiv Ereignisse aus einer Warteschlange oder einem Stream ab (Pull). Sie konfigurieren Lambda so, dass Lambda nach Ereignissen von einem unterstützten Service sucht, und Lambda kümmert sich um die Abfrage und den Aufruf Ihrer Funktion.

Wenn Ereignisse an Ihre Funktion übergeben werden, sind sie im JSON-Format strukturiert. Die JSON-Struktur variiert je nach Service, von dem sie generiert wird, und dem Ereignistyp. Während Lambda-Funktionsaufrufe standardmäßig bis zu 15 Minuten dauern können, eignet sich Lambda am besten für kurze Aufrufe, die eine Sekunde oder weniger dauern. Dies gilt insbesondere für ereignisgesteuerte Architekturen, bei denen jede Lambda-Funktion als Mikroservice behandelt wird, der für die Ausführung eines engen Satzes spezifischer Befehle verantwortlich ist.

> **Note**
>
> Event-driven Architekturen kommunizieren über verschiedene Systeme hinweg mithilfe von Netzwerken, was zu variabler Latenz führt. Für Workloads, die eine sehr geringe Latenz erfordern, wie z. B. Handelssysteme in Echtzeit, ist dieses Design möglicherweise nicht die beste Wahl. Für hoch skalierbare und verfügbare Workloads oder solche mit unvorhersehbaren Datenverkehrsmustern können ereignisgesteuerte Architekturen jedoch eine effektive Möglichkeit bieten, diese Anforderungen zu erfüllen.

Themen

- Vorteile einer ereignisgesteuerten Architektur
- Trade-offs ereignisgesteuerter Architekturen
- Anti-patterns in Lambda-based ereignisgesteuerten Anwendungen

Vorteile einer ereignisgesteuerten Architektur

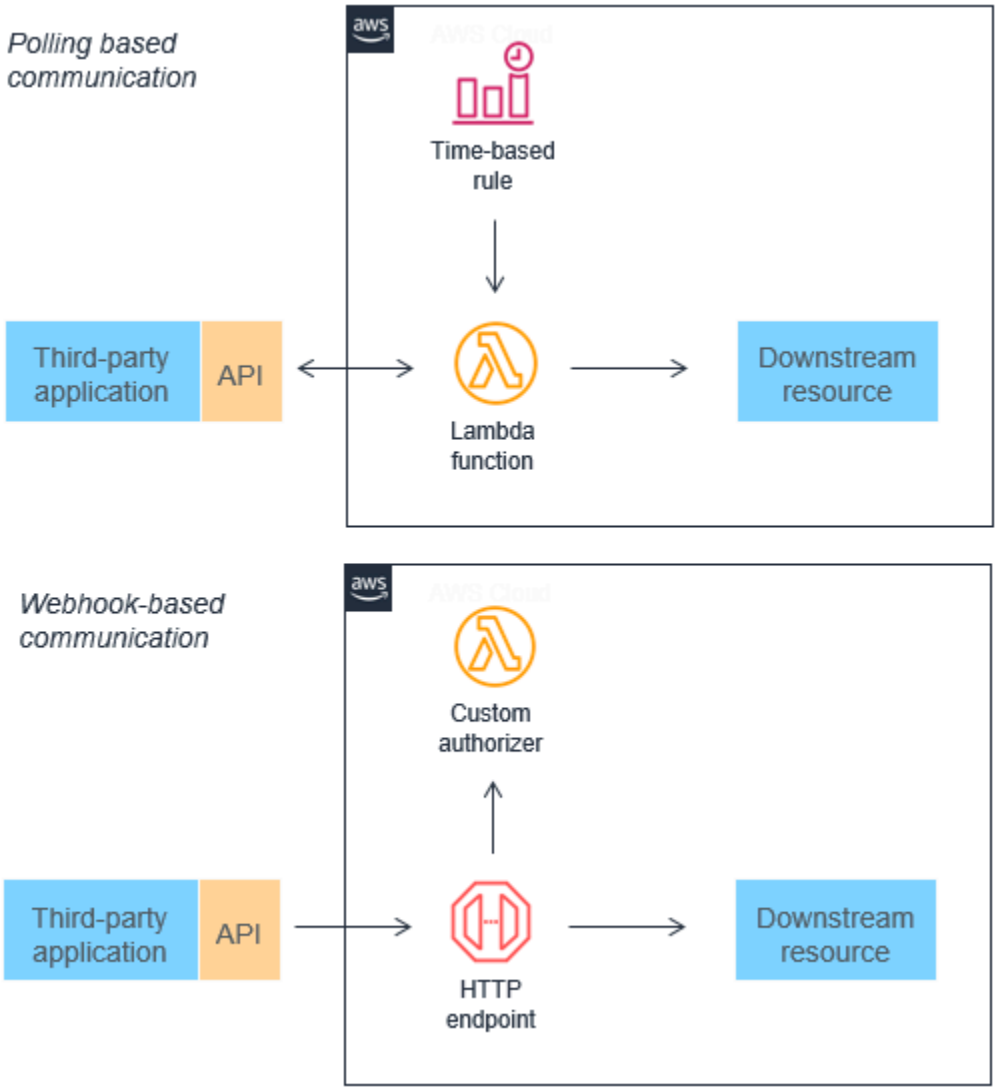
Lambda unterstützt zwei Aufrufmethoden in ereignisgesteuerten Architekturen:

1. Direkter Aufruf (Push-Methode): AWS Dienste lösen Lambda-Funktionen direkt aus. Beispiel:
   - Amazon S3 löst eine Funktion aus, wenn eine Datei hochgeladen wird.
   - API Gateway löst eine Funktion aus, wenn es eine HTTP-Anfrage empfängt.
2. Zuordnung von Ereignisquellen (Pull-Methode): Lambda ruft Ereignisse ab und ruft Funktionen auf. Beispiel:
   - Lambda ruft Nachrichten aus einer Amazon-SQS-Warteschlange ab und ruft eine Funktion auf.
   - Lambda liest Datensätze aus einem DynamoDB-Stream und ruft eine Funktion auf.

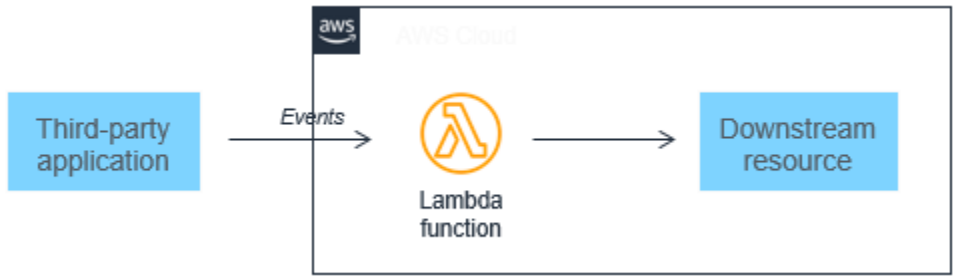
Beide Methoden tragen zu den Vorteilen von ereignisgesteuerten Architekturen bei, wie unten beschrieben.

Polling und Webhooks durch Ereignisse ersetzen

Viele traditionelle Architekturen verwenden Polling- und Webhook-Mechanismen, um den Status zwischen verschiedenen Komponenten zu kommunizieren. Das Abrufen von Aktualisierungen kann sehr ineffizient sein, da es eine Verzögerung zwischen der Verfügbarkeit neuer Daten und der Synchronisierung mit nachgelagerten Diensten gibt. Webhooks werden nicht immer von anderen Microservices unterstützt, in die Sie sich integrieren möchten. Sie erfordern möglicherweise auch benutzerdefinierte Autorisierungs- und Authentifizierungskonfigurationen. In beiden Fällen ist es schwierig, diese Integrationsmethoden ohne zusätzliche Arbeit der Entwicklungsteams bedarfsgerecht zu skalieren.



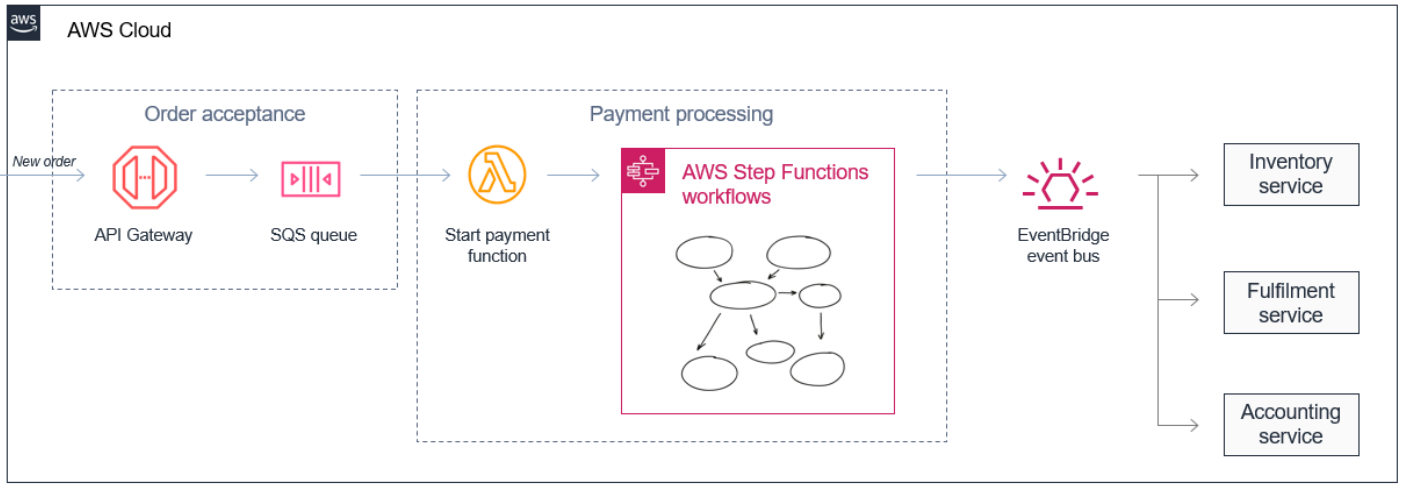
Diese beiden Mechanismen können durch Ereignisse ersetzt werden, die gefiltert, weitergeleitet und an nachgelagerte Microservices weitergeleitet werden können. Dieser Ansatz kann zu einem geringeren Bandbreitenverbrauch, einer geringeren CPU-Auslastung und potenziell niedrigeren Kosten führen. Diese Architekturen können auch die Komplexität verringern, da jede Funktionseinheit kleiner ist und oft weniger Code enthalten ist.



Event-driven Architekturen können es auch einfacher machen, Systeme zu entwerfen, die nahezu in Echtzeit laufen, und Unternehmen dabei unterstützen, sich von der stapelbasierten Verarbeitung zu verabschieden. Ereignisse werden zu dem Zeitpunkt erzeugt, an dem sich der Zustand der Anwendung ändert. Der benutzerdefinierte Code eines Microservices sollte daher so konzipiert sein, dass er die Verarbeitung eines einzelnen Ereignisses verarbeiten kann. Da die Skalierung durch den Lambda-Dienst erfolgt, kann diese Architektur einen erheblichen Anstieg des Datenverkehrs ohne Änderung des benutzerdefinierten Codes bewältigen. Mit der Zunahme von Ereignissen wächst auch die Datenverarbeitungsschicht, die die Ereignisse verarbeitet.

Reduzierung der Komplexität

Microservices ermöglichen es Entwicklern und Architekten, komplexe Arbeitsabläufe zu vereinfachen. Ein E-Commerce-Monolith kann beispielsweise in Auftragsannahme- und Zahlungsprozesse mit separaten Inventar-, Fulfillment- und Buchhaltungsdienstleistungen unterteilt werden. Was in einem Monolith komplex zu verwalten und zu orchestrieren sein könnte, wird zu einer Reihe von entkoppelten Diensten, die asynchron mit Ereignissen kommunizieren.



Dieser Ansatz ermöglicht auch die Zusammenstellung von Diensten, die Daten mit unterschiedlichen Geschwindigkeiten verarbeiten. In diesem Fall kann ein Microservice für die Auftragsannahme große Mengen eingehender Bestellungen speichern, indem er die Nachrichten in einer Amazon SQS SQS-Warteschlange zwischenspeichert.

Ein Zahlungsabwicklungsservice, der aufgrund der Komplexität der Zahlungsabwicklung in der Regel langsamer ist, kann einen stetigen Strom von Nachrichten aus der Amazon SQS-Warteschlange entgegennehmen. Er kann komplexe Wiederholungsversuche und Fehlerbehandlungslogik mithilfe von AWS Step Functions aktiven Zahlungsabläufen für Hunderttausende von Bestellungen orchestrieren und diese koordinieren.

Alternativer Ansatz: Für die Orchestrierung mit Standard-Programmiersprachen können Sie langlebige Lambda-Funktionen verwenden. Mit dauerhaften Funktionen können Sie die Auftragsannahme, die Zahlungsabwicklung und die Benachrichtigungslogik mit automatischem Checkpoint und automatischem Wiederholungsversuch in Code schreiben. Dieser Ansatz funktioniert gut, wenn der Workflow hauptsächlich Lambda-Funktionen umfasst und Sie es vorziehen, die Orchestrierungslogik im Code beizubehalten.

Verbesserung der Skalierbarkeit und Erweiterbarkeit

Microservices generieren Ereignisse, die in der Regel in Messaging-Diensten wie Amazon SNS und Amazon SQS veröffentlicht werden. Diese verhalten sich wie ein elastischer Puffer zwischen Microservices und helfen bei der Skalierung, wenn der Datenverkehr zunimmt. Dienste wie Amazon EventBridge können dann Nachrichten je nach Inhalt des Ereignisses filtern und weiterleiten, wie in den Regeln definiert. Infolgedessen sind ereignisbasierte Anwendungen skalierbarer und bieten eine größere Redundanz als monolithische Anwendungen.

Dieses System ist außerdem in hohem Maße erweiterbar, so dass andere Teams die Features erweitern und neue Features hinzufügen können, ohne dass dies Auswirkungen auf die Microservices für die Auftrags- und Zahlungsabwicklung hat. Durch die Veröffentlichung von Ereignissen mithilfe EventBridge dieser Anwendung lässt sie sich in bestehende Systeme wie den Inventar-Microservice integrieren, ermöglicht aber auch die Integration jeder future Anwendung als Event-Consumer. Die Produzenten von Ereignissen haben keine Kenntnis über die Konsumenten von Ereignissen, was zur Vereinfachung der Logik des Mikrodienstes beitragen kann.

Trade-offs ereignisgesteuerter Architekturen

Variable Latenzzeit

Im Gegensatz zu monolithischen Anwendungen, die möglicherweise alles innerhalb desselben Speicherbereichs auf einem einzigen Gerät verarbeiten, kommunizieren ereignisgesteuerte Anwendungen netzwerkübergreifend. Dieses Design führt zu einer variablen Latenzzeit. Es ist zwar möglich, Anwendungen so zu entwickeln, dass die Latenz minimiert wird, aber monolithische Anwendungen können fast immer für eine geringere Latenz optimiert werden, was auf Kosten der Skalierbarkeit und Verfügbarkeit geht.

Workloads, die eine konsistente Leistung mit geringer Latenz erfordern, wie z. B. Hochfrequenz-Handelsanwendungen in Banken oder die Submillisekunden-Automatisierung von Robotern in Lagerhäusern, sind keine guten Kandidaten für eine ereignisgesteuerte Architektur.

Letztendliche Datenkonsistenz

Ein Ereignis stellt eine Zustandsänderung dar und da zu einem bestimmten Zeitpunkt viele Ereignisse durch verschiedene Dienste in einer Architektur fließen, sind solche Workloads oft irgendwann konsistent. Dies erschwert die Verarbeitung von Transaktionen, den Umgang mit Duplikaten oder die Ermittlung des genauen Gesamtzustands eines Systems.

Einige Workloads enthalten eine Kombination von Anforderungen, die eventuell konsistent (z. B. Gesamtaufträge in der aktuellen Stunde) oder stark konsistent (z. B. aktueller Bestand) sind. Für Workloads, die eine hohe Datenkonsistenz erfordern, gibt es Architekturmuster, die dies unterstützen. Beispiel:

- DynamoDB kann strikt konsistente Lesevorgänge bereitstellen, manchmal mit einer höheren Latenz, wodurch ein höherer Durchsatz als im Standardmodus verbraucht wird. DynamoDB kann auch Transaktionen unterstützen, um die Datenkonsistenz aufrechtzuerhalten.
- Sie können Amazon RDS für Features verwenden, die ACID-Eigenschaften benötigen, obwohl relationale Datenbanken weniger skalierbar sind als ein NoSQL-Datenbanken wie DynamoDB. Amazon-RDS-Proxy kann dabei helfen, das Verbindungspooling und die Skalierung von kurzlebigen Verbrauchern wie Lambda-Funktionen zu verwalten.

Event-based Architekturen orientieren sich in der Regel an einzelnen Ereignissen und nicht an großen Datenmengen. Im Allgemeinen sind Workflows so konzipiert, dass sie die Schritte eines einzelnen Ereignisses oder Ausführungsablaufs verwalten, anstatt mehrere Ereignisse gleichzeitig zu bearbeiten. In Serverless-Systemen wird die Echtzeit-Ereignisverarbeitung der Batch-Verarbeitung vorgezogen: Batches sollten durch viele kleinere inkrementelle Aktualisierungen ersetzt werden. Dies kann zwar die Verfügbarkeit und Skalierbarkeit von Workloads verbessern, macht es aber auch schwieriger für Ereignisse, andere Ereignisse zu erkennen.

Rückgabe von Werten an Anrufer

In vielen Fällen sind ereignisbasierte Anwendungen asynchron. Das bedeutet, dass Anruferdienste nicht auf Anfragen von anderen Diensten warten, bevor sie mit anderen Aufgaben fortfahren. Dies ist eine grundlegende Eigenschaft ereignisgesteuerter Architekturen, die Skalierbarkeit und Flexibilität ermöglicht. Dies bedeutet, dass die Übergabe von Rückgabewerten oder des Ergebnisses eines Workflows komplexer ist als bei synchronen Ausführungsabläufen.

Die meisten Lambda-Aufrufe in Produktionssystemen sind asynchron und reagieren auf Ereignisse von Diensten wie Amazon S3 oder Amazon SQS. In diesen Fällen ist der Erfolg oder Misserfolg

der Verarbeitung eines Ereignisses oft wichtiger als die Rückgabe eines Wertes. Features wie Warteschlangen für unzustellbare Nachrichten (DLQs) in Lambda stellen sicher, dass Sie fehlgeschlagene Ereignisse identifizieren und erneut versuchen können, ohne den Aufrufer benachrichtigen zu müssen.

Dienst- und funktionsübergreifendes Debugging

Die Fehlersuche in ereignisgesteuerten Systemen unterscheidet sich ebenfalls von einer monolithischen Anwendung. Da verschiedene Systeme und Services Ereignisse weitergeben, ist es nicht möglich, den genauen Zustand mehrerer Services beim Auftreten von Fehlern zu erfassen und zu reproduzieren. Da jeder Dienst- und Funktionsaufruf über separate Protokolldateien verfügt, kann es komplizierter sein, festzustellen, was mit einem bestimmten Ereignis passiert ist, das einen Fehler verursacht hat.

Es gibt drei wichtige Voraussetzungen für den Aufbau eines erfolgreichen Debugging-Ansatzes in ereignisgesteuerten Systemen. Erstens ist ein robustes Protokollierungssystem von entscheidender Bedeutung. Dieses wird AWS dienstübergreifend bereitgestellt und von Amazon CloudWatch in Lambda-Funktionen eingebettet. Zweitens ist es in diesen Systemen wichtig, sicherzustellen, dass jedes Ereignis über eine Transaktions-ID verfügt, die bei jedem Schritt einer Transaktion protokolliert wird, um die Suche nach Protokollen zu erleichtern.

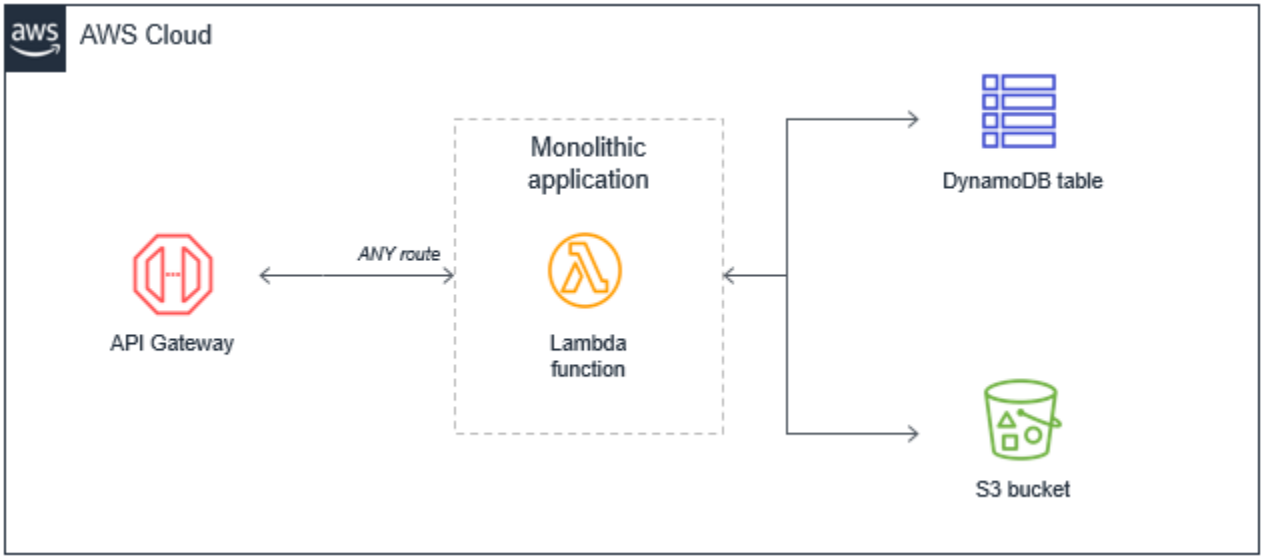
Schließlich wird dringend empfohlen, das Parsen und Analysieren von Protokollen mithilfe eines Debugging- und Überwachungsdienstes wie. AWS X-Ray Dies kann Protokolle über mehrere Lambda-Aufrufe und -Dienste hinweg verbrauchen, wodurch es viel einfacher wird, die Ursache von Problemen zu ermitteln. Ausführliche Informationen zur Verwendung X-Ray zur Fehlerbehebung finden Sie in der exemplarischen Vorgehensweise zur Fehlerbehebung.

Anti-patterns in Lambda-based ereignisgesteuerten Anwendungen

Vermeiden Sie beim Aufbau ereignisgesteuerter Architekturen mit Lambda die folgenden gängigen Anti-Patterns. Diese Muster funktionieren, können aber die Kosten und die Komplexität erhöhen.

Der Lambda-Monolith

In vielen Anwendungen, die von herkömmlichen Servern migriert wurden, wie z. B. Amazon EC2 EC2-Instances oder Elastic Beanstalk Beanstalk-Anwendungen, müssen Entwickler vorhandenen Code „nach Belieben verschieben“. Dies führt häufig zu einer einzigen Lambda-Funktion, die die gesamte Anwendungslogik enthält, die für alle Ereignisse ausgelöst wird. Bei einer einfachen Webanwendung würde eine monolithische Lambda-Funktion alle API-Gateway-Routen verwalten und mit allen erforderlichen nachgelagerten Ressourcen integrieren.

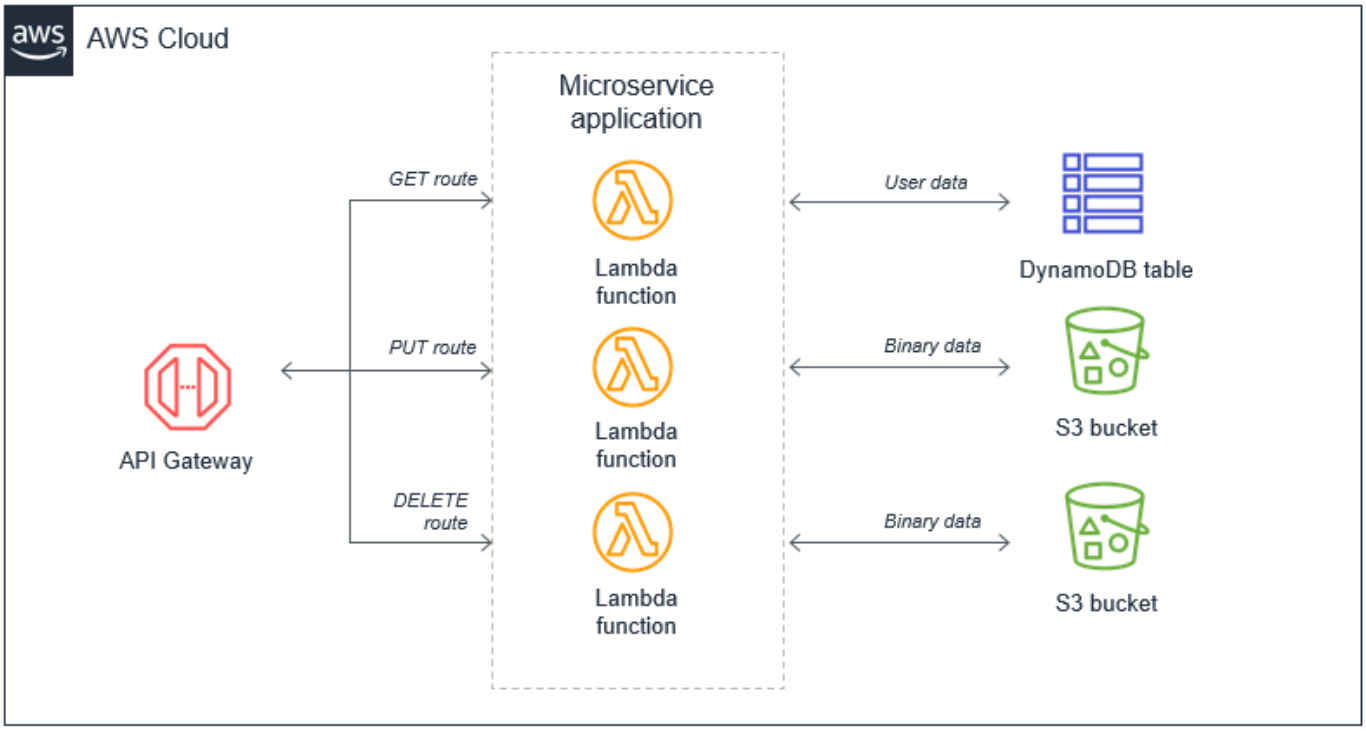


Dieser Ansatz hat mehrere Nachteile:

- Paketgröße — Die Lambda-Funktion ist möglicherweise viel größer, da sie den gesamten möglichen Code für alle Pfade enthält, wodurch die Ausführung des Lambda-Dienstes langsamer wird.
- Es ist schwierig, die geringsten Rechte durchzusetzen — Die Ausführungsrolle der Funktion muss Berechtigungen für alle Ressourcen gewähren, die für alle Pfade benötigt werden, sodass die Berechtigungen sehr weit gefasst sind. Dies ist ein Sicherheitsproblem. Viele Pfade im funktionalen Monolithen benötigen nicht alle erteilten Berechtigungen.
- Schwieriger zu aktualisieren: In einem Produktionssystem sind Upgrades für eine einzelne Funktion riskanter und können dazu führen, dass die gesamte Anwendung nicht mehr funktioniert. Das Upgrade eines einzelnen Pfads in der Lambda-Funktion ist ein Upgrade der gesamten Funktion.
- Schwieriger zu warten — Es ist schwieriger, mehrere Entwickler an dem Service arbeiten zu lassen, da es sich um ein monolithisches Code-Repository handelt. Es erhöht auch die kognitive Belastung der Entwickler und erschwert es, eine angemessene Testabdeckung für Code zu schaffen.
- Schwieriger, Code wiederzuverwenden: Es kann schwieriger sein, wiederverwendbare Bibliotheken von Monolithen zu trennen, was die Wiederverwendung von Code erschwert. Je mehr Projekte Sie entwickeln und unterstützen, desto schwieriger wird es, den Code zu unterstützen und die Geschwindigkeit Ihres Teams zu erhöhen.
- Schwieriger zu testen: Mit zunehmender Anzahl von Codezeilen wird es immer schwieriger, für alle möglichen Kombinationen von Eingaben und Einstiegspunkten in der Codebasis einen Modultest

durchzuführen. Im Allgemeinen ist es einfacher, Unit-Tests für kleinere Dienste mit weniger Code zu implementieren.

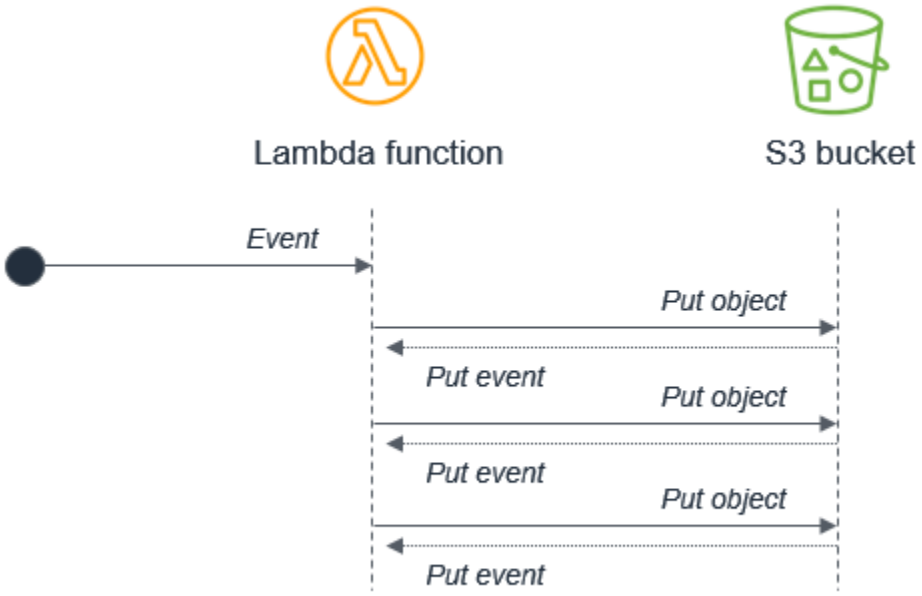
Die bevorzugte Alternative besteht darin, die monolithische Lambda-Funktion in einzelne Microservices aufzuteilen und eine einzelne Lambda-Funktion einer einzigen, klar definierten Aufgabe zuzuordnen. In dieser einfachen Webanwendung mit einigen API-Endpunkten kann die resultierende Microservice-basierte Architektur auf den API-Gateway-Routen basieren.



Rekursive Muster, die außer Kontrolle geratene Lambda-Funktionen verursachen

AWS Dienste generieren Ereignisse, die Lambda-Funktionen aufrufen, und Lambda-Funktionen können Nachrichten an Dienste senden. AWS Im Allgemeinen sollte sich der Dienst oder die Ressource, die eine Lambda-Funktion aufruft, von dem Dienst oder der Ressource unterscheiden, an den die Funktion ausgibt. Wenn dies nicht verwaltet wird, kann dies zu Endlosschleifen führen.

Beispielsweise schreibt eine Lambda-Funktion ein Objekt in ein Amazon S3 S3-Objekt, das wiederum dieselbe Lambda-Funktion mithilfe eines Put-Ereignisses aufruft. Durch den Aufruf wird ein zweites Objekt in den Bucket geschrieben, das dieselbe Lambda-Funktion aufruft:



Während das Potenzial für Endlosschleifen in den meisten Programmiersprachen vorhanden ist, hat dieses Anti-Pattern das Potenzial, mehr Ressourcen in Serverless-Anwendungen zu verbrauchen. Sowohl Lambda als auch Amazon S3 skalieren automatisch auf der Grundlage des Datenverkehrs, sodass die Schleife dazu führen kann, dass Lambda skaliert, um die gesamte verfügbare Parallelität zu verbrauchen, und Amazon S3 weiterhin Objekte schreibt und mehr Ereignisse für Lambda generiert.

In diesem Beispiel wird S3 verwendet, aber das Risiko von rekursiven Schleifen besteht auch bei Amazon SNS, Amazon SQS, DynamoDB und anderen Diensten. Sie können die rekursive Schleifenerkennung verwenden, um dieses Anti-Pattern zu finden und zu vermeiden.

Lambda-Funktionen, die Lambda-Funktionen aufrufen

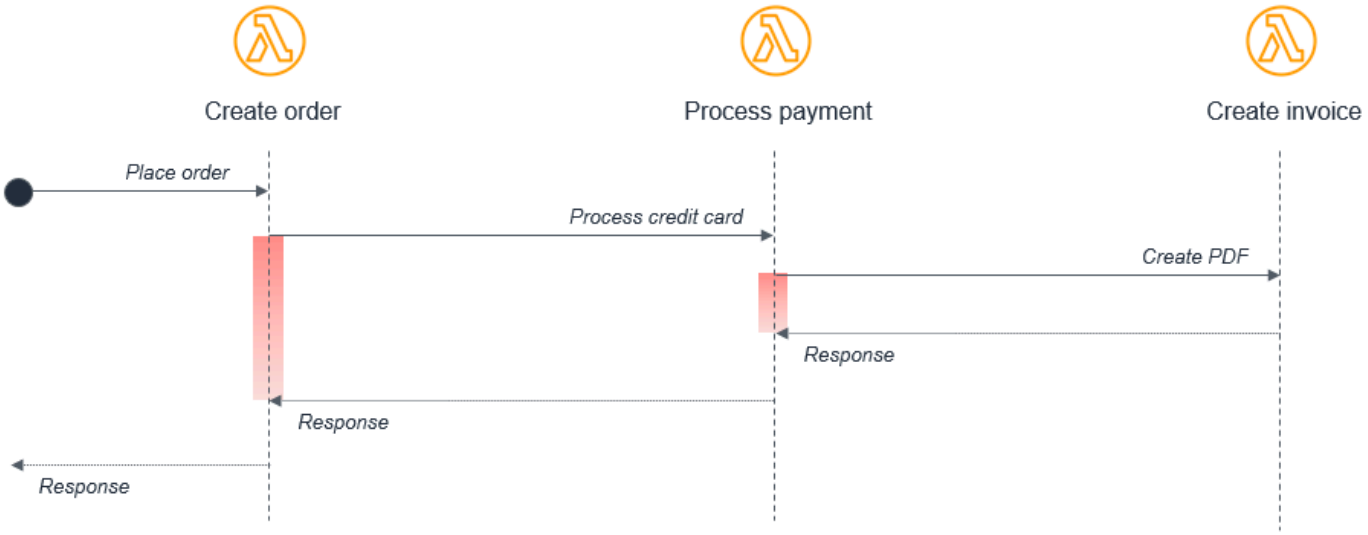
Funktionen ermöglichen die Kapselung und die Wiederverwendung von Code. Die meisten Programmiersprachen unterstützen das Konzept des synchronen Aufrufs von Funktionen innerhalb einer Codebasis. Wenn die Funktion einen Fehler zurückgibt, gibt die Funktion eine Antwort zurück.

> **Note**
>
> Während Lambda-Funktionen, die direkt andere Lambda-Funktionen aufrufen, aus Kosten- und Komplexitätsgründen im Allgemeinen ein Anti-Pattern darstellen, gilt dies nicht

für langlebige Funktionen, die speziell dafür konzipiert sind, mehrstufige Workflows zu orchestrieren, indem sie andere Funktionen aufrufen.

Wenn dies auf einem herkömmlichen Server oder einer virtuellen Instance geschieht, wechselt der Scheduler des Betriebssystems zu einer anderen verfügbaren Arbeit. Ob die CPU zu 0 % oder zu 100 % läuft, hat keinen Einfluss auf die Gesamtkosten der Anwendung, da Sie für die Fixkosten des Besitzes und des Betriebs eines Servers zahlen.

Dieses Modell eignet sich häufig nicht gut für die Serverless-Entwicklung. Nehmen wir zum Beispiel eine einfache E-Commerce-Anwendung, die aus drei Lambda-Funktionen besteht, die eine Bestellung verarbeiten:



In diesem Fall ruft die Funktion Bestellung erstellen die Funktion Zahlung bearbeiten auf, die wiederum die Funktion Rechnung erstellen aufruft. Dieser synchrone Ablauf kann zwar innerhalb einer einzelnen Anwendung auf einem Server funktionieren, führt jedoch zu mehreren vermeidbaren Problemen in einer verteilten serverlosen Architektur:

- Kosten: Bei Lambda zahlen Sie für die Dauer eines Aufrufs. In diesem Beispiel laufen während der Ausführung der Funktion Rechnung erstellen zwei weitere Funktionen in einem Wartezustand, der im Diagramm in Rot dargestellt ist.
- Fehlerbehandlung: Bei verschachtelten Aufrufen kann die Fehlerbehandlung viel komplexer werden. Ein Fehler in Rechnung erstellen kann beispielsweise dazu führen, dass die Funktion „Zahlung verarbeiten“ die Belastung rückgängig macht, oder es wird stattdessen der Vorgang „Rechnung erstellen“ erneut versucht.

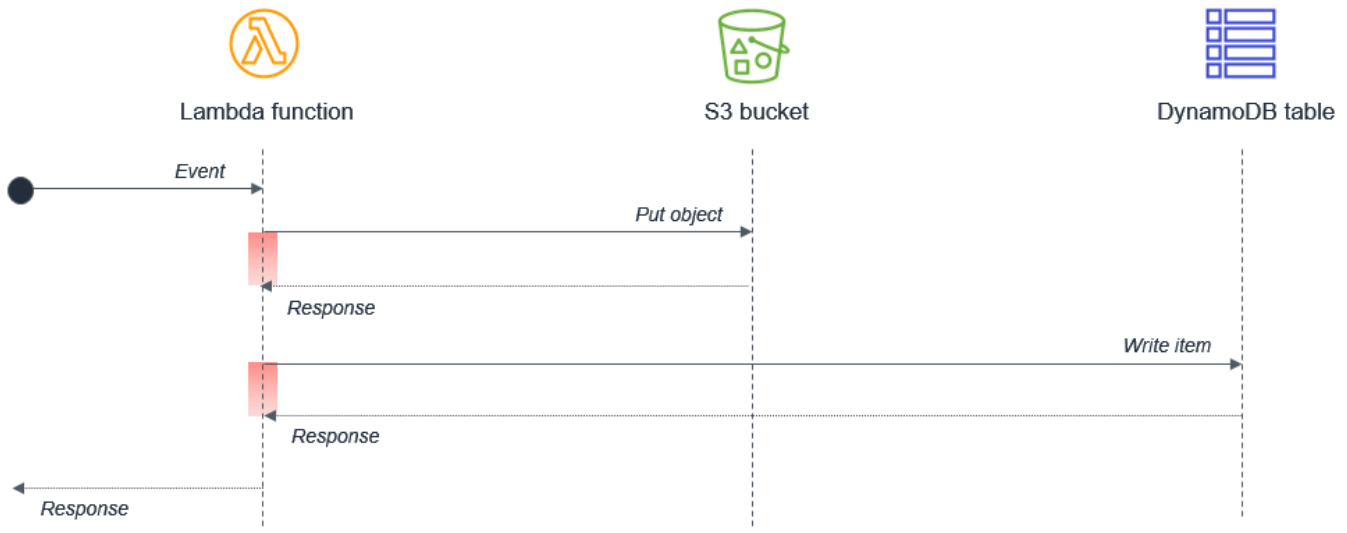
- Enge Verknüpfung: Die Bearbeitung einer Zahlung dauert in der Regel länger als die Erstellung einer Rechnung. Bei diesem Modell wird die Verfügbarkeit des gesamten Workflows durch die langsamste Funktion eingeschränkt.
- Skalierung: Die Gleichzeitigkeit aller drei Funktionen muss gleich sein. In einem stark frequentierten System wird dadurch mehr Gleichzeitigkeit verwendet, als andernfalls erforderlich wäre.

Bei Serverless-Anwendungen gibt es zwei gängige Ansätze, um dieses Muster zu vermeiden. Verwenden Sie zunächst eine Amazon-SQS-Warteschlange zwischen Lambda-Funktionen. Wenn ein nachgeschalteter Prozess langsamer ist als ein vorgeschalteter Prozess, hält die Warteschlange die Nachrichten dauerhaft fest und entkoppelt die beiden Funktionen. In diesem Beispiel würde die Funktion Bestellung erstellen eine Nachricht in einer Amazon SQS SQS-Warteschlange veröffentlichen, und die Funktion Zahlung verarbeiten verarbeitet Nachrichten aus der Warteschlange.

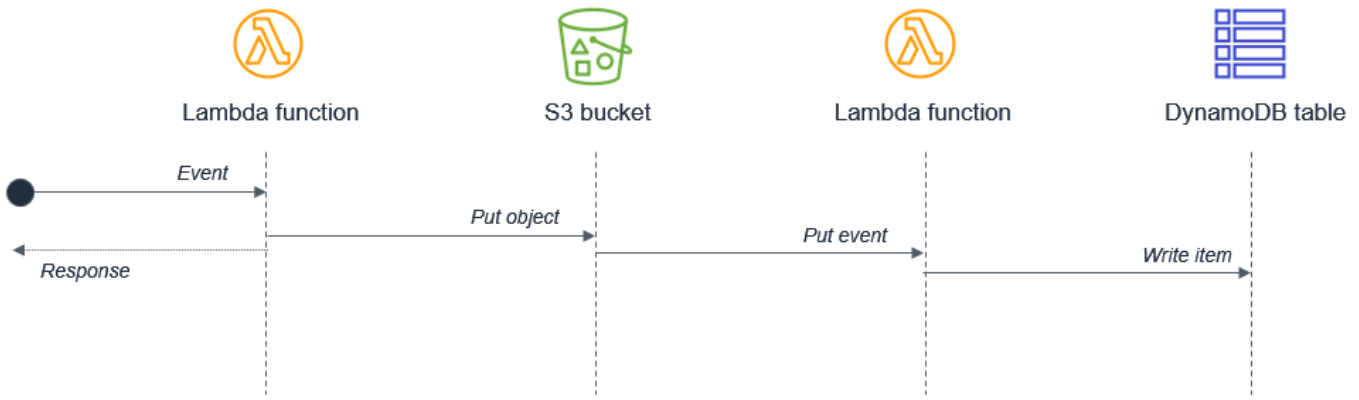
Der zweite Ansatz besteht darin, AWS Step Functions zu verwenden. Bei komplexen Prozessen mit mehreren Arten von Fehlern und Wiederholungslogik können Schrittfunktionen dazu beitragen, den Umfang des benutzerdefinierten Codes zu reduzieren, der für die Orchestrierung des Workflows erforderlich ist. Infolgedessen orchestriert Step Functions die Arbeit und behandelt Fehler und Wiederholungen zuverlässig und die Lambda-Funktionen enthalten nur Geschäftslogik.

Synchrones Warten innerhalb einer einzigen Lambda-Funktion

Stellen Sie sicher, dass potenziell gleichzeitige Aktivitäten nicht synchron innerhalb einer einzelnen Lambda-Funktion geplant werden. Eine Lambda-Funktion könnte beispielsweise in einen S3-Bucket und dann in eine DynamoDB-Tabelle schreiben:



In diesem Design sind die Wartezeiten zusammengesetzt, weil die Aktivitäten sequentiell sind. In Fällen, in denen die zweite Aufgabe von der Fertigstellung der ersten Aufgabe abhängt, können Sie die Gesamtwartezeit und die Ausführungskosten verringern, indem Sie zwei separate Lambda-Funktionen verwenden:



In diesem Design reagiert die erste Lambda-Funktion sofort, nachdem das Objekt in den Amazon-S3-Bucket gestellt wurde. Der S3-Dienst ruft die zweite Lambda-Funktion auf, die dann Daten in die DynamoDB-Tabelle schreibt. Dieser Ansatz minimiert die Gesamtwartezeit bei der Ausführung von Lambda-Funktionen.

Entwerfen von Lambda-Anwendungen

Eine gut konzipierte ereignisgesteuerte Anwendung verwendet eine Kombination aus AWS Diensten und benutzerdefiniertem Code, um Anfragen und Daten zu verarbeiten und zu verwalten. Dieses Kapitel konzentriert sich auf Lambda-specific Themen des Anwendungsdesigns. Bei der Entwicklung von Anwendungen für stark frequentierte Produktionssysteme gibt es viele wichtige Überlegungen für Serverless-Architekten.

Viele der bewährten Verfahren, die für die Softwareentwicklung und verteilte Systeme gelten, gelten auch für die Entwicklung Serverless-Anwendungen. Das übergeordnete Ziel besteht darin, Workloads zu entwickeln, die:

- zuverlässig sind: Sie bieten Ihren Endnutzern ein hohes Maß an Verfügbarkeit. AWS -Serverless-Dienste sind zuverlässig, weil sie auch für Ausfälle ausgelegt sind.
- langlebig sind: Sie bieten Speicheroptionen, die den Anforderungen an die Haltbarkeit Ihrer Workloads gerecht werden.
- sicher sind: Befolgen Sie bewährte Verfahren und verwenden Sie die bereitgestellten Tools, um den Zugriff auf Workloads zu sichern und den Explosionsradius zu begrenzen.
- leistungsstark sind: effiziente Nutzung von Rechenressourcen und Erfüllung der Leistungsanforderungen Ihrer Endbenutzer.
- Cost-efficient— Entwicklung von Architekturen, die unnötige Kosten vermeiden, die ohne Mehrausgaben skaliert werden können und die auch ohne nennenswerten Mehraufwand außer Betrieb genommen werden können.

Die folgenden Designprinzipien können Ihnen dabei helfen, Workloads zu erstellen, die diese Ziele erfüllen. Möglicherweise gelten nicht alle Prinzipien für jede Architektur, aber sie sollten Ihnen bei allgemeinen Architekturentscheidungen als Leitfaden dienen.

Themen

- Verwendung von Diensten anstelle von benutzerdefiniertem Code
- Informationen zu Lambda-Abstraktionsebenen
- Implementieren der Zustandslosigkeit in Funktionen
- Kopplung minimieren
- Entwickeln für On-Demand-Daten statt Batches
- Wählen Sie eine Orchestrierungsoption für komplexe Workflows

- Implementieren von Idempotenz
- Verwenden Sie mehrere AWS Konten für die Verwaltung von Kontingenten

Verwendung von Diensten anstelle von benutzerdefiniertem Code

Serverlose Anwendungen bestehen normalerweise aus mehreren AWS Diensten, die mit benutzerdefiniertem Code integriert sind, der in Lambda-Funktionen ausgeführt wird. Lambda kann zwar in die meisten AWS Dienste integriert werden, aber die Dienste, die in serverlosen Anwendungen am häufigsten verwendet werden, sind:

Kategorie	AWS Dienst
Datenverarbeitung	AWS Lambda
Datenspeicher	Amazon S3 Amazon DynamoDB Amazon RDS
API	Amazon API Gateway
Integration von Anwendungen	Amazon EventBridge Amazon SNS Amazon SQS
Orchestrierung	Langlebige Lambda-Funktionen AWS Step Functions
Streaming-Daten und Analysen	Amazon Data Firehose

> **Note**
>
> Viele Serverless-Dienste bieten Replikation und Support für mehrere Regionen, einschließlich DynamoDB und Amazon S3. Lambda-Funktionen können als Teil einer Bereitstellungspipeline in mehreren Regionen bereitgestellt werden und API Gateway

kann so konfiguriert werden, dass es diese Konfiguration unterstützt. Sehen Sie sich diese Beispielarchitektur an, die zeigt, wie dies erreicht werden kann.

Es gibt viele etablierte, gängige Muster in verteilten Architekturen, die Sie selbst erstellen oder mithilfe von Diensten implementieren können. AWS Für die meisten Kunden ist es wirtschaftlich nicht sinnvoll, Zeit in die Entwicklung dieser Muster zu investieren. Wenn Ihre Anwendung eines dieser Muster benötigt, verwenden Sie den entsprechenden AWS Dienst:

Muster	AWS Dienst
Warteschlange	Amazon SQS
Ereignisbus	Amazon EventBridge
Publish/subscribe (Fanout)	Amazon SNS
Orchestrierung	Langlebige Lambda-Funktionen AWS Step Functions
API	Amazon API Gateway
Ereignis-Streams	Amazon Kinesis

Diese Dienste sind für die Integration mit Lambda konzipiert und Sie können Infrastructure as Code (IaC) verwenden, um Ressourcen in den Diensten zu erstellen und zu verwerfen. Sie können jeden dieser Dienste über das AWS SDK nutzen, ohne Anwendungen installieren oder Server konfigurieren zu müssen. Die Fähigkeit, diese Dienste mithilfe von Code in Ihren Lambda-Funktionen zu nutzen, ist ein wichtiger Schritt zur Entwicklung gut gestalteter serverloser Anwendungen.

Informationen zu Lambda-Abstraktionsebenen

Der Lambda-Dienst beschränkt Ihren Zugriff auf die zugrunde liegenden Betriebssysteme, Hypervisoren und Hardware, auf denen Ihre Lambda-Funktionen ausgeführt werden. Der Dienst wird ständig verbessert und die Infrastruktur verändert, um neue Features hinzuzufügen, die Kosten zu senken und den Dienst leistungsfähiger zu machen. Ihr Code sollte kein Wissen darüber voraussetzen, wie Lambda aufgebaut ist und keine Hardware-Affinität voraussetzen.

In ähnlicher Weise werden die Integrationen von Lambda mit anderen Diensten von verwaltet AWS, sodass Ihnen nur eine geringe Anzahl von Konfigurationsoptionen zur Verfügung steht. Bei der Interaktion zwischen API Gateway und Lambda gibt es beispielsweise kein Konzept für den Load Balancer, da dieser vollständig von den Diensten verwaltet wird. Sie haben auch keine direkte Kontrolle darüber, welche Verfügbarkeitszonen die Dienste beim Aufrufen von Funktionen zu einem bestimmten Zeitpunkt verwenden oder wie Lambda entscheidet, wann die Anzahl der Ausführungsumgebungen nach oben oder unten skaliert wird.

Diese Abstraktion ermöglicht es Ihnen, sich auf die Integrationsaspekte Ihrer Anwendung, den Datenfluss und die Geschäftslogik zu konzentrieren, bei denen Ihr Workload einen Mehrwert für Ihre Endbenutzer bietet. Wenn Sie den Diensten die Verwaltung der zugrundeliegenden Mechanismen überlassen, können Sie Anwendungen schneller entwickeln und müssen weniger eigenen Code pflegen.

Implementieren der Zustandslosigkeit in Funktionen

Bei Lambda-Funktionen sollten Sie davon ausgehen, dass die Umgebung nur für einen einzigen Aufruf existiert. Die Funktion sollte beim ersten Start alle erforderlichen Zustände initialisieren. Beispielsweise erfordert Ihre Funktion möglicherweise das Abrufen von Daten aus einer DynamoDB-Tabelle. Sie sollte alle dauerhaften Datenänderungen vor dem Beenden in einem dauerhaften Speicher wie Amazon S3, DynamoDB oder Amazon SQS speichern. Sie sollte sich nicht auf bestehende Datenstrukturen oder temporäre Dateien oder auf interne Zustände stützen, die bei mehreren Aufrufen verwaltet würden.

Um Datenbankverbindungen und Bibliotheken zu initialisieren oder den Status zu laden, können Sie die statische Initialisierung nutzen. Da Ausführungsumgebungen nach Möglichkeit wiederverwendet werden, um die Leistung zu verbessern, können Sie die für die Initialisierung dieser Ressourcen benötigte Zeit über mehrere Aufrufe hinweg amortisieren. Sie sollten jedoch keine Variablen oder Daten, die in der Funktion verwendet werden, in diesem globalen Bereich speichern.

Kopplung minimieren

Die meisten Architekturen sollten viele, kürzere Funktionen den wenigen, größeren vorziehen. Der Zweck jeder Funktion sollte darin bestehen, das an die Funktion übergebene Ereignis zu verarbeiten, ohne Kenntnisse oder Erwartungen in Bezug auf den gesamten Arbeitsablauf oder das Transaktionsvolumen. Dadurch ist die Funktion unabhängig von der Quelle des Ereignisses und nur minimal mit anderen Diensten verbunden.

Alle Konstanten mit globalem Geltungsbereich, die sich nur selten ändern, sollten als Umgebungsvariablen implementiert werden, um Aktualisierungen ohne Bereitstellungen zu ermöglichen. Alle Geheimnisse oder sensiblen Informationen sollten im AWS Systems Manager Parameter Store oder AWS Secrets Manager gespeichert und von der Funktion geladen werden. Da diese Ressourcen kontospezifisch sind, können Sie auf diese Weise Build-Pipelines für mehrere Konten erstellen. Die Pipelines laden die entsprechenden Geheimnisse für jede Umgebung, ohne dass diese für Entwickler sichtbar sind oder Codeänderungen erforderlich sind.

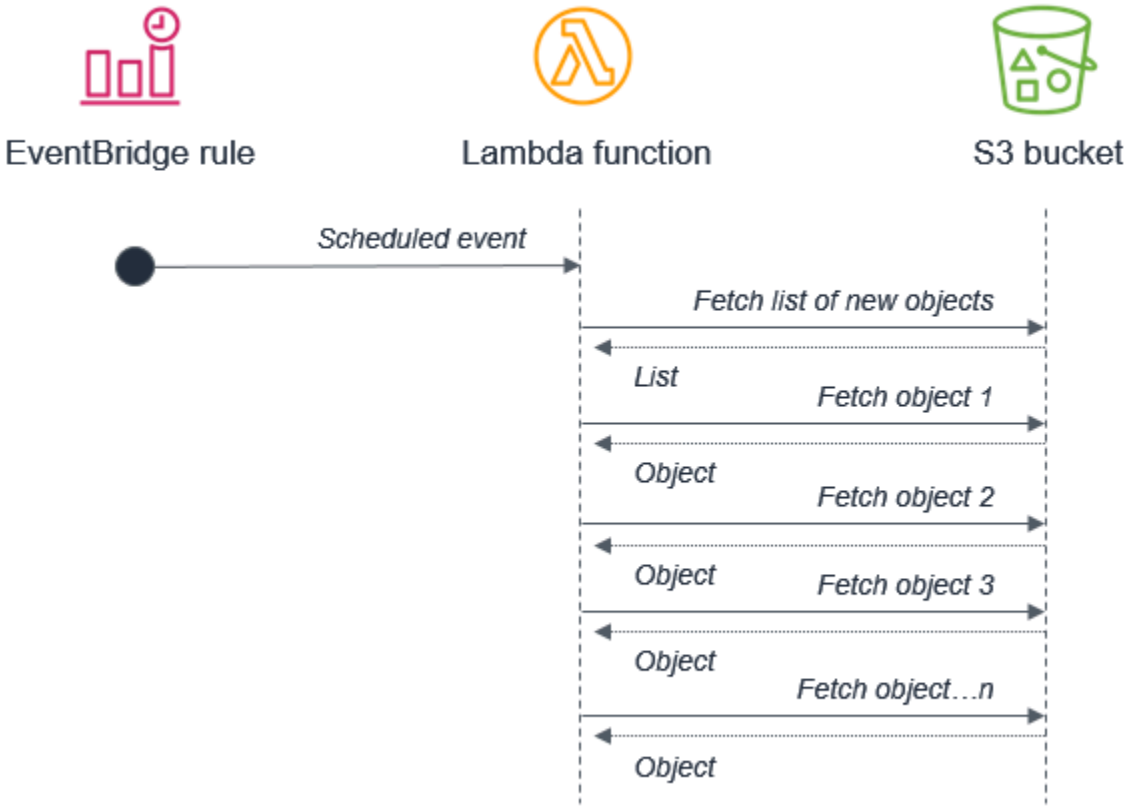
Entwickeln für On-Demand-Daten statt Batches

Viele herkömmliche Systeme sind so konzipiert, dass sie periodisch laufen und Batch von Transaktionen verarbeiten, die sich im Laufe der Zeit angesammelt haben. Beispielsweise kann eine Bankanwendung stündlich ausgeführt werden, um Geldautomaten-Transaktionen in zentrale Hauptbücher zu verarbeiten. In Lambda-based Anwendungen sollte die benutzerdefinierte Verarbeitung bei jedem Ereignis ausgelöst werden, sodass der Dienst die Parallelität nach Bedarf skalieren kann, sodass Transaktionen nahezu in Echtzeit verarbeitet werden können.

Standardmäßig kann ein Funktionsaufruf bis zu 15 Minuten lang ausgeführt werden. Auf Lambda Managed Instances können asynchrone Aufrufe und ESM-Aufrufe (Event Source Mapping) bis zu 90 Minuten lang ausgeführt werden. Bei länger andauernder Verarbeitung überprüfen langlebige Funktionen den Fortschritt über mehrere Aufrufe hinweg. Bei asynchronem Aufruf kann eine dauerhafte Ausführung bis zu einem Jahr dauern. Dennoch sollten Sie, wenn möglich, die ereignisgesteuerte Verarbeitung der Stapelverarbeitung vorziehen.

Sie können Cron-Aufgaben zwar in serverlosen Anwendungen ausführen, indem Sie geplante Ausdrücke für Regeln in Amazon verwenden EventBridge, diese sollten jedoch sparsam oder als letztes Mittel verwendet werden. Bei jeder geplanten Aufgabe, die einen Batch verarbeitet, besteht die Möglichkeit, dass das Transaktionsvolumen über das hinausgeht, was innerhalb der Lambda-Dauerbegrenzung verarbeitet werden kann. Wenn Sie aufgrund der Einschränkungen externer Systeme gezwungen sind, einen Scheduler zu verwenden, sollten Sie in der Regel für den kürzesten sinnvollen wiederkehrenden Zeitraum planen.

Es ist beispielsweise nicht empfehlenswert, einen Batch-Prozess zu verwenden, der eine Lambda-Funktion auslöst, um eine Liste neuer Amazon S3 S3-Objekte abzurufen. Dies liegt daran, dass der Dienst zwischen den Batches möglicherweise mehr neue Objekte empfängt, als innerhalb der Lambda-Dauerbegrenzung verarbeitet werden können.



Stattdessen sollte Amazon S3 die Lambda-Funktion jedes Mal aufrufen, wenn ein neues Objekt in den Bucket gestellt wird. Dieser Ansatz ist deutlich skalierbarer und funktioniert nahezu in Echtzeit.



Wählen Sie eine Orchestrierungsoption für komplexe Workflows

Workflows, die Verzweigungslogik, verschiedene Arten von Fehlermodellen und Wiederholungslogik beinhalten, verwenden in der Regel einen Orchestrator, um den Status der Gesamtausführung zu verfolgen. Erstellen Sie keine Ad-hoc-Orchestrierung in Lambda-Funktionen. Dies führt zu einer engen Kopplung, komplexem Routing-Code und keiner automatischen Statuswiederherstellung.

Verwenden Sie stattdessen eine dieser speziell entwickelten Orchestrierungsoptionen:

- Dauerhafte Lambda-Funktionen: Application-centric Orchestrierung mithilfe von Standardprogrammiersprachen mit automatischem Checkpointing, integriertem

Wiederholungsversuch und Ausführungswiederherstellung. Ideal für Entwickler, die es vorziehen, die Workflow-Logik im Code neben der Geschäftslogik in Lambda beizubehalten.

- AWS Step Functions: Visuelle Workflow-Orchestrierung mit nativen Integrationen für mehr als 220 Dienste. AWS Ideal für die Koordination mehrerer Dienste, eine wartungsfreie Infrastruktur und ein visuelles Workflow-Design.

Anleitungen zur Auswahl zwischen diesen Optionen finden Sie unter Dauerhafte Funktionen oder Step Functions.

Mit Step Functions verwenden Sie Zustandsmaschinen, um die Orchestrierung zu verwalten. Dadurch werden die Fehlerbehandlung, das Routing und die Verzweigungslogik aus Ihrem Code extrahiert und durch Zustandsautomaten ersetzt, die mit JSON deklariert werden. Abgesehen davon, dass es Workflows robuster und beobachtbarer macht, ermöglicht es Ihnen auch, Workflows zu versionieren und den Zustandsautomaten zu einer kodifizierten Ressource zu machen, die Sie zu einem Code-Repository hinzufügen können.

Es ist üblich, dass einfachere Workflows in Lambda-Funktionen mit der Zeit komplexer werden. Beim Betrieb einer serverlosen Produktionsanwendung ist es wichtig, zu erkennen, wann dies geschieht, damit Sie diese Logik auf eine Zustandsmaschine oder eine dauerhafte Funktion migrieren können.

Implementieren von Idempotenz

AWS Serverlose Dienste, einschließlich Lambda, sind fehlertolerant und für den Umgang mit Ausfällen konzipiert. Wenn beispielsweise ein Dienst eine Lambda-Funktion aufruft und es zu einer Dienstunterbrechung kommt, ruft Lambda Ihre Funktion in einer anderen Verfügbarkeitszone auf. Wenn Ihre Funktion einen Fehler auslöst, wiederholt Lambda den Aufruf.

Da dasselbe Ereignis mehr als einmal empfangen werden kann, sollten Funktionen so konzipiert sein, dass sie idempotent sind. Dies bedeutet, dass der mehrfache Empfang desselben Ereignisses das Ergebnis nicht über den ersten Empfang hinaus verändert.

Sie können Idempotenz in Lambda-Funktionen implementieren, indem Sie eine DynamoDB-Tabelle verwenden, um kürzlich verarbeitete Bezeichner zu verfolgen und festzustellen, ob die Transaktion bereits zuvor bearbeitet wurde. Die DynamoDB-Tabelle implementiert in der Regel einen TTL-Wert (Time To Live) für ablaufende Elemente, um den verwendeten Speicherplatz zu begrenzen.

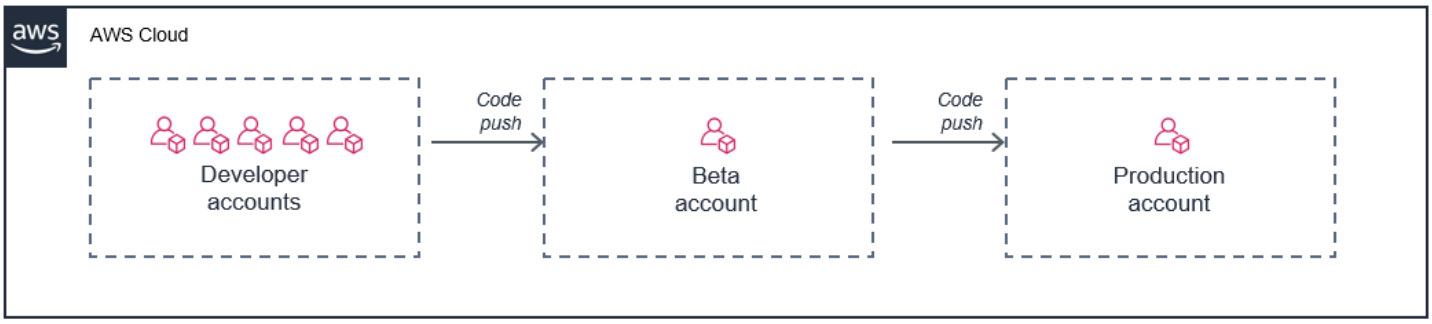
Verwenden Sie mehrere AWS Konten für die Verwaltung von Kontingenten

Viele Servicekontingente AWS werden auf Kontoebene festgelegt. Das bedeutet, dass Sie bei zunehmendem Workload schnell an Ihre Grenzen stoßen können.

Eine effektive Möglichkeit, dieses Problem zu lösen, besteht darin, mehrere AWS Konten zu verwenden und jede Arbeitslast einem eigenen Konto zuzuweisen. Dadurch wird verhindert, dass Kontingente mit anderen Workloads oder Ressourcen außerhalb der Produktion geteilt werden.

Mithilfe von AWS -Organisationen können Sie außerdem die Abrechnung, Compliance und Sicherheit dieser Konten zentral verwalten. Sie können Richtlinien an Gruppen von Konten anhängen, um benutzerdefinierte Skripts und manuelle Prozesse zu vermeiden.

Ein gängiger Ansatz besteht darin, jedem Entwickler ein AWS Konto zur Verfügung zu stellen und dann separate Konten für die Betabereitstellungsphase und die Produktion zu verwenden:



In diesem Modell hat jeder Entwickler seine eigenen Beschränkungen für das Konto, sodass sich deren Nutzung nicht auf Ihre Produktionsumgebung auswirkt. Dieser Ansatz ermöglicht es Entwicklern auch, Lambda-Funktionen lokal auf ihren Entwicklungsmaschinen anhand von Live-Cloud-Ressourcen in ihren individuellen Konten zu testen.

Erstellen Sie Ihre erste Lambda-Funktion

für Ihre ersten Schritte mit Lambda verwenden Sie die Lambda-Konsole, um eine Funktion zu erstellen. In wenigen Minuten können Sie eine Funktion erstellen und bereitstellen und sie in der Konsole testen.

Bei der Durchführung des Tutorials lernen Sie einige grundlegende Lambda-Konzepte kennen, z. B. wie Sie Argumente an Ihre Funktion mithilfe des Lambda-Ereignisobjekts übergeben. Sie erfahren auch, wie Sie Protokollausgaben Ihrer Funktion zurückgeben und wie Sie die Aufrufprotokolle Ihrer Funktion in Amazon anzeigen. CloudWatch Protokolle.

Um die Dinge einfach zu halten, erstellen Sie Ihre Funktion entweder mit Python oder Node.js Laufzeit. Mit diesen interpretierten Sprachen können Sie Funktionscode direkt im integrierten Code-Editor der Konsole bearbeiten. Bei kompilierten Sprachen wie Java und C# müssen Sie ein Bereitstellungspaket auf Ihrem lokalen Build-Rechner erstellen und es in Lambda hochladen. Weitere Informationen zum Bereitstellen von Funktionen in Lambda mithilfe anderer Laufzeiten finden Sie unter den Links im Abschnitt the section called “Nächste Schritte”.

> **Tip**
>
> Weitere Informationen zum Erstellen von Serverless-Lösungen finden Sie im Serverless-Benutzerhandbuch.

Melde dich an für AWS-Konto

Um loszulegen mit AWS, du brauchst eine AWS-Konto. Es gibt zwei Möglichkeiten, sich anzumelden AWS. Wenn Sie die vollständige Kontrolle über die Kontokonfiguration und den Speicherort Ihrer Inhalte haben möchten oder wenn Sie regulierte Workloads wie HIPAA oder FedRAMP haben, verwenden Sie Melden Sie sich an für AWS (fortgeschritten). Wenn Sie ein neuer Benutzer sind und ein Konto verwenden möchten, das Sie bereits besitzen, um sich anzumelden AWS und erstellen Sie eine AWS Umgebung mit vorkonfigurierten Standardwerten in select AWS-Regionen, verwenden Melde dich an für AWS.

Erstellen einer Lambda-Funktion mit der Konsole

In diesem Beispiel besitzt Ihre Funktion ein JSON-Objekt, das zwei ganzzahlige Werte mit der Bezeichnung `"length"` und `"width"` enthält. Die Funktion multipliziert diese Werte, um eine Fläche zu berechnen, und gibt diese als JSON-Zeichenfolge zurück.

Ihre Funktion druckt auch die berechnete Fläche zusammen mit dem Namen ihrer CloudWatch Gruppe protokollieren. Später im Tutorial lernen Sie, wie Sie es verwenden CloudWatch Logs um Aufzeichnungen über den Aufruf Ihrer Funktionen einzusehen.

So erstellen Sie eine Hello-World-Lambda-Funktion mit der Konsole

1. Öffnen Sie das Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie Event-Funktion erstellen.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie im Bereich Grundlegende Informationen als Funktionsname **`myLambdaFunction`** ein.
5. Für Laufzeit, wähle entweder Node.js 24 oder Python 3.14.
6. Verlassen Architektur gesetzt auf x86_64, und wählen Sie dann Event-Funktion erstellen.

Zusätzlich zu einer einfachen Funktion, die die Nachricht `Hello from Lambda!` zurückgibt, erstellt Lambda auch eine Ausführungsrolle für Ihre Funktion. Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM) -Rolle, die einer Lambda-Funktion die Zugriffsberechtigung erteilt AWS-Services und Ressourcen. Für Ihre Funktion gewährt die Rolle, die Lambda erstellt, grundlegende Schreibberechtigungen CloudWatch Protokolle.

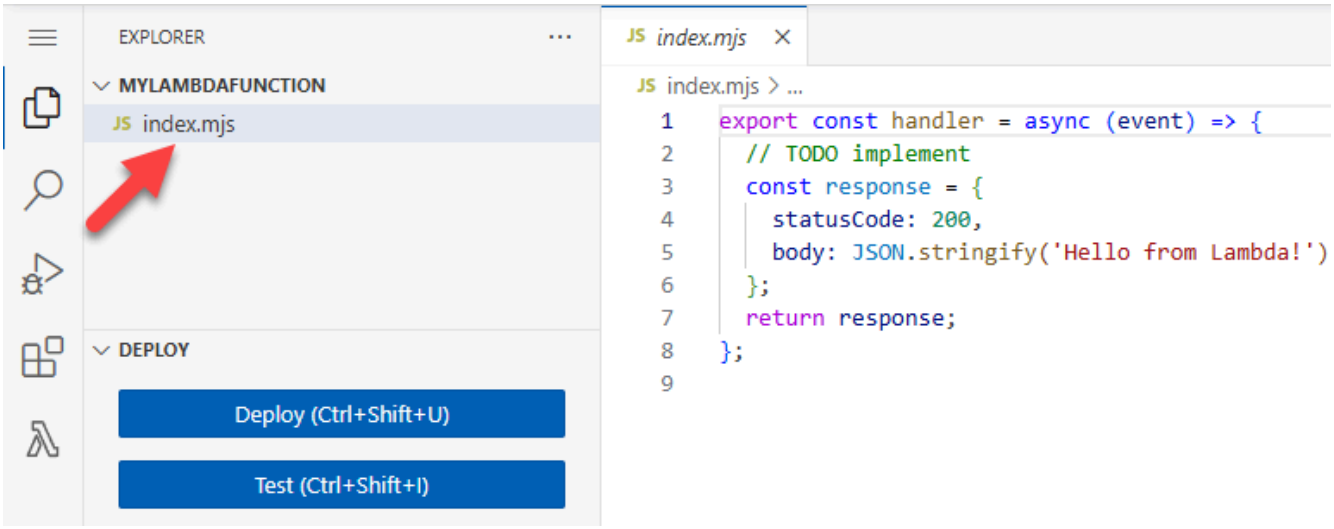
Verwenden Sie den integrierten Code-Editor der Konsole, um den von Lambda erstellten Hello-World-Code durch Ihren eigenen Funktionscode zu ersetzen.

Node.js

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

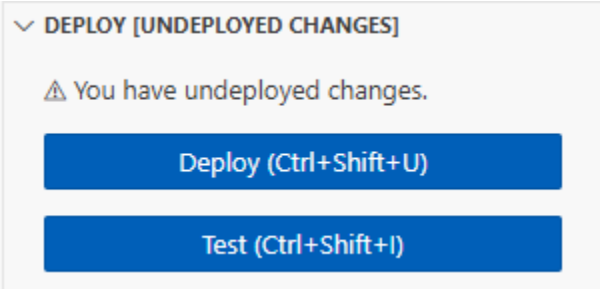
   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte index.mjs im Code-Editor nicht angezeigt wird, wählen Sie index.mjs im Datei-Explorer aus, wie im folgenden Diagramm gezeigt.



2. Fügen Sie den folgenden Code in die Registerkarte index.mjs ein und ersetzen Sie den von Lambda erstellten Code.

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Den Funktionscode verstehen

Bevor Sie zum nächsten Schritt übergehen, sollten Sie sich einen Moment Zeit nehmen, um den Funktionscode zu betrachten und einige wichtige Lambda-Konzepte zu verstehen.

- Der Lambda-Handler:

  Ihre Lambda-Funktion enthält eine Node.js Funktion benannt `handler`. Eine Lambda-Funktion in Node.js kann mehr als eine enthalten Node.js Funktion, aber Handler Die Funktion ist immer der Einstiegspunkt zu Ihrem Code. Wenn Ihre Funktion aufgerufen wird, führt Lambda diese Methode aus.

  Wenn Sie Ihre Hello-World-Funktion über die Konsole erstellt haben, hat Lambda den Namen der Handler-Methode für Ihre Funktion automatisch auf `handler` festgelegt. Achten Sie darauf, den Namen nicht zu bearbeiten Node.js Funktion. Wenn Sie dies tun, kann Lambda Ihren Code nicht ausführen, wenn Sie Ihre Funktion aufrufen.

  Weitere Informationen zum Lambda-Handler finden Sie unter Node.js, siehe the section called “Handler”.
- Das Lambda-Ereignisobjekt:

  Die Funktion `handler` besitzt zwei Argumente, `event` und `context`. Ein Ereignis in Lambda ist ein JSON-formatiertes Dokument, das Daten enthält, die von Ihrer Funktion verarbeitet werden sollen.

  Wenn Ihre Funktion von einem anderen aufgerufen wird AWS-Service, das Event-Objekt enthält Informationen über das Ereignis, das den Aufruf verursacht hat. Wenn Ihre Funktion zum Beispiel aufgerufen wird, wenn ein Objekt in einen Amazon Simple Storage Service (Amazon S3) Bucket hochgeladen wird, enthält das Ereignis den Namen des Buckets und den Objektschlüssel.

In diesem Beispiel erstellen Sie ein Ereignis in der Konsole, indem Sie ein JSON-formatiertes Dokument mit zwei Schlüssel-Wert-Paaren eingeben.

- Das Lambda-Kontextobjekt:

  Das zweite Argument, das Ihre Funktion besitzt, ist `context`. Lambda übergibt das Kontextobjekt automatisch an Ihre Funktion. Das Kontextobjekt enthält Informationen über den Aufruf der Funktion und die Ausführungsumgebung.

  Mit dem Kontextobjekt können Sie zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion ausgeben. In diesem Beispiel verwendet Ihre Funktion `logGroupName` Parameter zur Ausgabe des Namens seiner CloudWatch Protokollgruppe.

  Weitere Informationen zum Lambda-Kontextobjekt finden Sie unter Node.js, siehe the section called “Kontext”.

- Protokollierung in Lambda:

  Mit Node.js, Sie können Konsolenmethoden verwenden wie `console.log` und `console.error` um Informationen an das Protokoll Ihrer Funktion zu senden. Der Beispielcode verwendet `console.log` Anweisungen zur Ausgabe der berechneten Fläche und des Namens der Funktion CloudWatch Gruppe „Protokolle“. Sie können auch jede Protokollierungsbibliothek verwenden, die in `stdout` oder `stderr` schreibt.

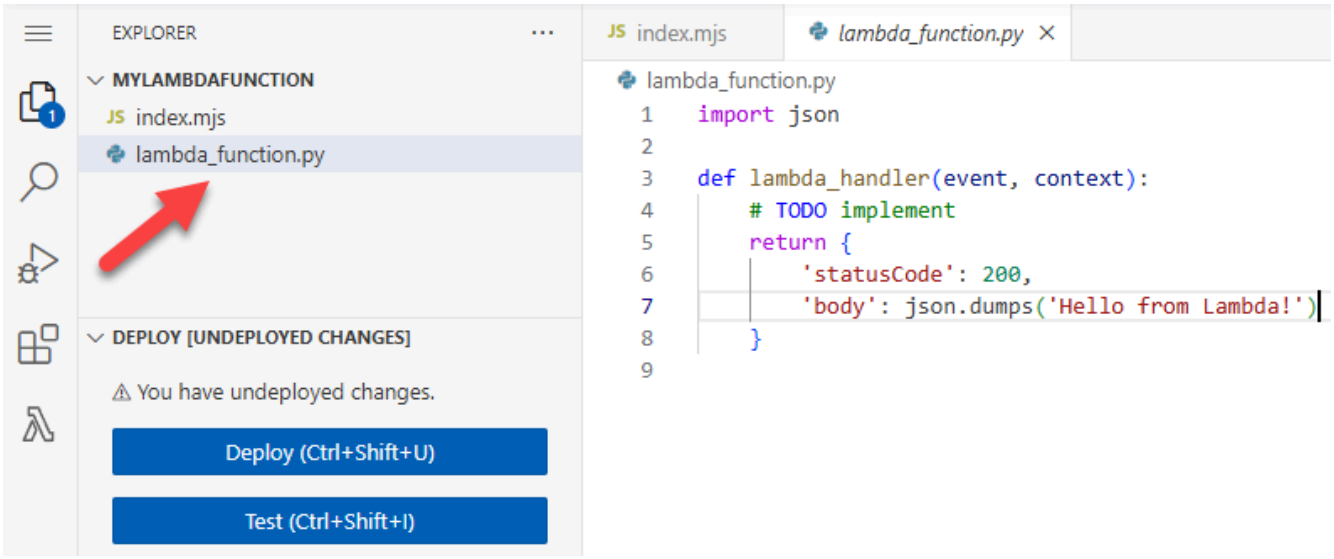
  Weitere Informationen hierzu finden Sie unter the section called “Protokollierung”. Informationen zum Protokollieren in anderen Laufzeiten finden Sie auf den Seiten „Erstellen mit“ für die Laufzeiten, an denen Sie interessiert sind.

Python

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

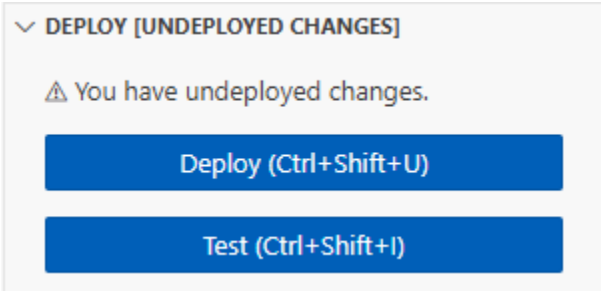
   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte lambda_function.py im Code-Editor nicht angezeigt wird, wählen Sie lambda_function.py im Datei-Explorer aus, wie im folgenden Diagramm dargestellt.



2. Fügen Sie den folgenden Code in die Registerkarte lambda_function.py ein und ersetzen Sie den von Lambda erstellten Code.

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
    width = event['width']

    area = calculate_area(length, width)
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Den Funktionscode verstehen

Bevor Sie zum nächsten Schritt übergehen, sollten Sie sich einen Moment Zeit nehmen, um den Funktionscode zu betrachten und einige wichtige Lambda-Konzepte zu verstehen.

- Der Lambda-Handler:

  Ihre Lambda-Funktion enthält eine Python-Funktion mit dem Namen `lambda_handler`. Eine Lambda-Funktion in Python kann mehr als eine Python-Funktion enthalten, aber die Handler-Funktion ist immer der Einstiegspunkt in Ihren Code. Wenn Ihre Funktion aufgerufen wird, führt Lambda diese Methode aus.

  Wenn Sie Ihre Hello-World-Funktion über die Konsole erstellt haben, hat Lambda den Namen der Handler-Methode für Ihre Funktion automatisch auf `lambda_handler` festgelegt. Achten Sie darauf, den Namen dieser Python-Funktion nicht zu bearbeiten. Wenn Sie dies tun, kann Lambda Ihren Code nicht ausführen, wenn Sie Ihre Funktion aufrufen.

  Weitere Informationen zum Lambda-Handler in Python finden Sie unter the section called “Handler”.

- Das Lambda-Ereignisobjekt:

  Die Funktion `lambda_handler` besitzt zwei Argumente, `event` und `context`. Ein Ereignis in Lambda ist ein JSON-formatiertes Dokument, das Daten enthält, die von Ihrer Funktion verarbeitet werden sollen.

  Wenn Ihre Funktion von einer anderen aufgerufen wird AWS-Service, das Event-Objekt enthält Informationen über das Ereignis, das den Aufruf verursacht hat. Wenn Ihre Funktion zum Beispiel aufgerufen wird, wenn ein Objekt in einen Amazon Simple Storage Service

(Amazon S3) Bucket hochgeladen wird, enthält das Ereignis den Namen des Buckets und den Objektschlüssel.

In diesem Beispiel erstellen Sie ein Ereignis in der Konsole, indem Sie ein JSON-formatiertes Dokument mit zwei Schlüssel-Wert-Paaren eingeben.

- Das Lambda-Kontextobjekt:

  Das zweite Argument, das Ihre Funktion besitzt, ist `context`. Lambda übergibt das Kontextobjekt automatisch an Ihre Funktion. Das Kontextobjekt enthält Informationen über den Aufruf der Funktion und die Ausführungsumgebung.

  Mit dem Kontextobjekt können Sie zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion ausgeben. In diesem Beispiel verwendet Ihre Funktion `log_group_name` Parameter zur Ausgabe des Namens seiner CloudWatch Protokollgruppe.

  Weitere Informationen zum Lambda-Kontextobjekt in Python finden Sie unter the section called “Kontext”.

- Protokollierung in Lambda:

  Mit Python können Sie entweder eine `print`-Anweisung oder eine Python-Protokollbibliothek verwenden, um Informationen an das Protokoll Ihrer Funktion zu senden. Um den Unterschied in der Erfassung zu veranschaulichen, werden im Beispielcode beide Methoden verwendet. In einer Produktionsanwendung empfehlen wir die Verwendung einer Protokollierungsbibliothek.

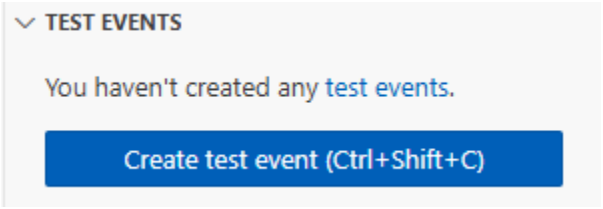
  Weitere Informationen hierzu finden Sie unter the section called “Protokollierung”. Informationen zum Protokollieren in anderen Laufzeiten finden Sie auf den Seiten „Erstellen mit“ für die Laufzeiten, an denen Sie interessiert sind.

Aufrufen der Lambda-Funktion mithilfe des Konsolencode-Editors

Um Ihre Funktion mit dem Code-Editor der Lambda-Konsole aufzurufen, erstellen Sie ein Testereignis, das Sie an Ihre Funktion senden. Beim Ereignis handelt es sich um ein JSON-formatiertes Dokument, das zwei Schlüssel-Wert-Paare mit den Schlüsseln `"length"` und `"width"`. enthält.

So erstellen Sie das Testereignis

1. Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolen-Code-Editors die Option Testereignis erstellen aus.



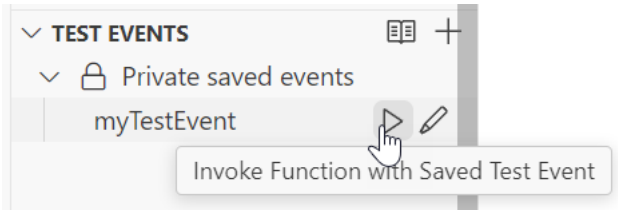
2. Geben Sie als Event name (Ereignisname) die Zeichenfolge **`myTestEvent`** ein.
3. Ersetzen Sie im Abschnitt Event JSON das Standard-JSON durch Folgendes:

```
{
  "length": 6,
  "width": 7
}
```

4. Wählen Sie Speichern.

So testen Sie Ihre Funktion und zeigen Aufrufe an

Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolencode-Editors das Ausführungssymbol neben Ihrem Testereignis:



Wenn Ihre Funktion beendet ist, werden die Antwort- und Funktionsprotokolle auf der Registerkarte OUTPUT angezeigt. Sie sollten ähnliche Ergebnisse wie nachfolgend zu sehen:

Node.js

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\":42}"
```

```
Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2024-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is 42
2024-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch log
 group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

Python

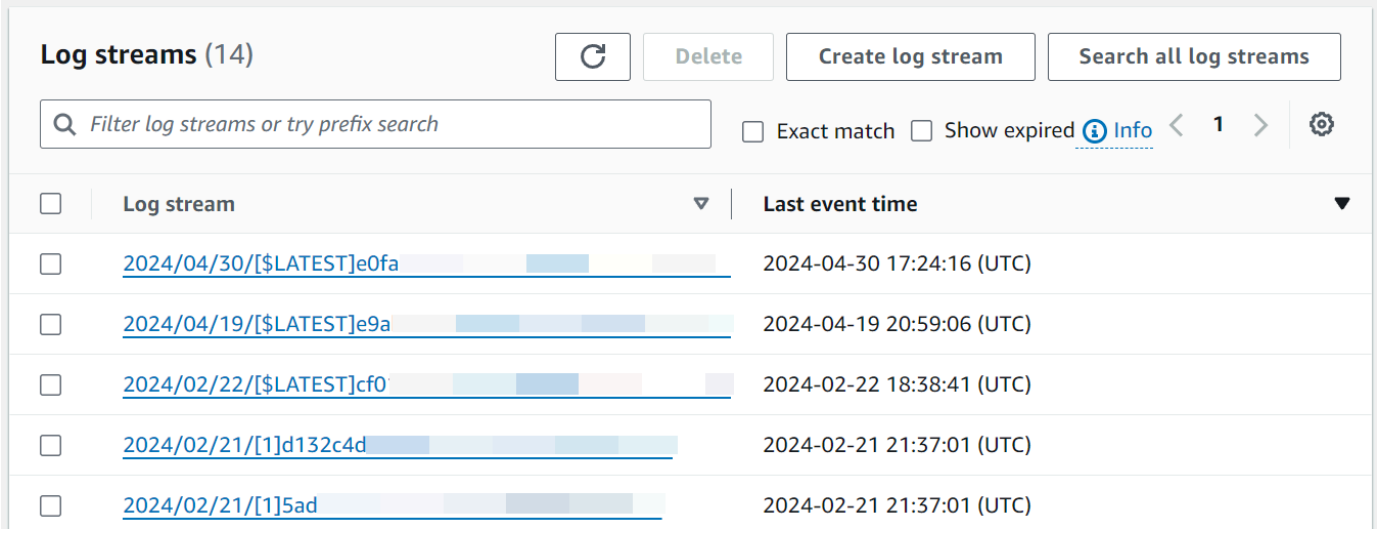
```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2024-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch logs
 group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

Wenn Sie Ihre Funktion außerhalb der Lambda-Konsole aufrufen, müssen Sie CloudWatch Protokolle, um die Ausführungsergebnisse Ihrer Funktion anzuzeigen.

Um die Aufrufdatensätze Ihrer Funktion einzusehen in CloudWatch Logs

1. Öffnen Sie das Gruppen protokollieren Seite der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/myLambdaFunction`) aus. Dies ist der Name der Protokollgruppe, den Ihre Funktion an die Konsole ausgegeben hat.

3. Scrollen Sie nach unten und wählen Sie den Protokollstream für die Funktionsaufrufen aus, die Sie sich ansehen möchten.



Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

Node.js

```
INIT_START Runtime Version: nodejs:22.v13    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
2024-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
 is 42
2024-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
 CloudWatch log group:  /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
 Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
 Duration: 178.85 ms
```

Python

```
INIT_START Runtime Version: python:3.13.v16    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2024-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
```

```
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

Bereinigen

Wenn Sie mit der Beispielfunktion fertig sind, löschen Sie sie. Sie können auch die Protokollgruppe löschen, in der die Protokolle der Funktion gespeichert sind, sowie die von der Konsole erstellte Ausführungsrolle.

Um die Lambda-Event-Funktion zu löschen

1. Öffnen Sie das Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie die Event-Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Protokollgruppe

1. Öffnen Sie die Seite Log Groups (Protokollgruppen) der CloudWatch-Konsole.
2. Wählen Sie die Protokollgruppe der Funktion (`/aws/lambda/myLambdaFunction`).
3. Wählen Sie Actions (Aktionen), Delete log group(s) (Protokollgruppe(n) löschen) aus.
4. Wählen Sie im Dialogfeld Delete log group(s) (Protokollgruppe(n) löschen) die Option Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie das Seite „Rollen“ der AWS Identity and Access Management (IAM) -Konsole.
2. Wählen Sie die Ausführungsrolle der Funktion aus (zum Beispiel `myLambdaFunction-role-`*`31exxmpl`*).
3. Wählen Sie Löschen aus.
4. Geben Sie im Dialogfenster Delete role (Rolle löschen) den Namen der Rolle ein und wählen Sie anschließend Delete (Löschen) aus.

Zusätzliche Ressourcen und nächste Schritte

Nachdem Sie nun eine einfache Lambda-Funktion mithilfe der Konsole erstellt und getestet haben, gehen Sie wie folgt vor:

- Erfahren Sie, wie Sie Ihrer Funktion Abhängigkeiten hinzufügen und diesen mithilfe eines ZIP-Bereitstellungspakets bereitstellen. Wählen Sie aus den folgenden Links Ihre bevorzugte Sprache aus.

  Node.js

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Typescript

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Python

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Ruby

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Java

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Go

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  C#

  the section called “Bereitstellungspaket”

- Um zu erfahren, wie Sie eine Lambda-Funktion mit einer anderen aufrufen AWS-Service, siehe Tutorial: Verwenden eines Amazon S3 S3-Triggers zum Aufrufen einer Lambda-Ereignisfunktion.
- Wählen Sie eines der folgenden Tutorials für komplexere Beispiele für die Verwendung von Lambda mit anderen AWS-Services.
  - Tutorial: Verwenden von Lambda mit API Gateway: Erstellen Sie eine Amazon-API-Gateway-REST-API, die eine Lambda-Funktion aufruft.

- Verwenden einer Lambda-Funktion für den Zugriff auf eine Amazon-RDS-Datenbank: Verwenden Sie eine Lambda-Funktion, um Daten über den RDS-Proxy in eine Datenbank des Amazon Relational Database Service (Amazon RDS) zu schreiben.
- Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern: Verwenden Sie eine Lambda-Funktion, um bei jedem Hochladen einer Image-Datei in einen Amazon-S3-Bucket ein Miniaturbild zu erstellen.

Erste Schritte mit Beispielanwendungen und -mustern

Die folgenden Ressourcen können verwendet werden, um schnell Serverless-Apps zu erstellen und bereitzustellen, die einige gängige Lambda-Anwendungsfälle implementieren. Für jede der Beispiel-Apps stellen wir Anweisungen zur Verfügung, um Ressourcen entweder manuell mit dem AWS-Managementkonsole zu erstellen und zu konfigurieren oder AWS Serverless Application Model um die Ressourcen mithilfe von IaC bereitzustellen. Folgen Sie den Anweisungen auf der Konsole, um mehr über die Konfiguration der einzelnen AWS Ressourcen für jede App zu erfahren, oder verwenden Sie sie, AWS SAM um Ressourcen schnell bereitzustellen, wie Sie es in einer Produktionsumgebung tun würden.

Dateiverarbeitung

- PDF-Verschlüsselungsanwendung: Erstellen Sie eine Serverless-Anwendung, die PDF-Dateien beim Hochladen in einen Bucket in Amazon Simple Storage Service verschlüsselt und in einem anderen Bucket speichert. Dies ist nützlich, um vertrauliche Dokumente beim Hochladen zu schützen.

- Bildanalyseanwendung: Erstellen Sie eine Serverless-Anwendung, die mithilfe von Amazon Rekognition Text aus Bildern extrahiert. Dies ist nützlich für die Dokumentverarbeitung, Inhaltsmoderation und automatisierte Bildanalyse.

Datenbankintegration

- Queue-to-Database Anwendung : Erstellen Sie eine serverlose Anwendung, die Warteschlangennachrichten in eine Amazon RDS-Datenbank schreibt. Dies ist nützlich für die Verarbeitung von Benutzerregistrierungen und die Bearbeitung von Bestellungen.

- Datenbankereignis-Handler: Erstellen Sie eine Serverless-Anwendung, die auf Änderungen in Amazon-DynamoDB-Tabellen reagiert. Dies ist nützlich für Audit-Protokollierung, Datenreplikation und automatisierte Workflows.

Geplante Aufgaben

- Anwendung zur Datenbankwartung : Erstellen Sie eine serverlose Anwendung, die automatisch Einträge, die älter als 12 Monate sind, nach einem Cron-Zeitplan aus einer Amazon DynamoDB-Tabelle löscht. Dies ist nützlich für die automatische Datenbankwartung und das Datenlebenszyklusmanagement.
- Erstellen Sie einen EventBridge Scheduler-Zeitplan für Lambda-Funktionen : Verwenden Sie den EventBridge Scheduler, um eine Lambda-Funktion nach einem einmaligen oder wiederkehrenden Zeitplan aufzurufen. EventBridge Der Scheduler unterstützt Cron- und Rate-Ausdrücke, flexible Zeitfenster, Wiederholungsrichtlinien und Warteschlangen, in denen keine Nachrichten mehr angezeigt werden. Sie können einen Zeitplan direkt von der Lambda-Konsole aus mithilfe des Add-Trigger-Flows oder von der Scheduler-Konsole aus erstellen. EventBridge

Weitere Ressourcen

Die folgenden Ressourcen enthalten weitere Informationen zu Lambda und zur Entwicklung von Serverless-Anwendungen:

- Serverless Land: Eine Bibliothek mit gebrauchsfertigen Mustern für die Erstellung von Serverless-Apps. Es hilft Entwicklern, Anwendungen mithilfe von AWS Diensten wie Lambda, API Gateway und schneller zu erstellen. EventBridge Die Website bietet vorgefertigte Lösungen und bewährte Methoden, die die Entwicklung von Serverless-Systemen erleichtern.
- Lambda-Beispielanwendungen : Anwendungen, die im GitHub Repository für dieses Handbuch verfügbar sind. Diese Beispiele veranschaulichen die Verwendung verschiedener Sprachen und AWS Dienste. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung und unterstützende Ressourcen.
- Codebeispiele für Lambda mithilfe von AWS SDKs : Beispiele, die Ihnen zeigen, wie Sie Lambda mit AWS Software Development Kits (SDKs) verwenden. Diese Beispiele umfassen Grundlagen, Aktionen, Szenarien und Beiträge aus der Community. AWS Die Beispiele umfassen grundlegende Operationen, einzelne Servicefunktionen und spezifische Aufgaben, bei denen mehrere Funktionen oder AWS Dienste verwendet werden.

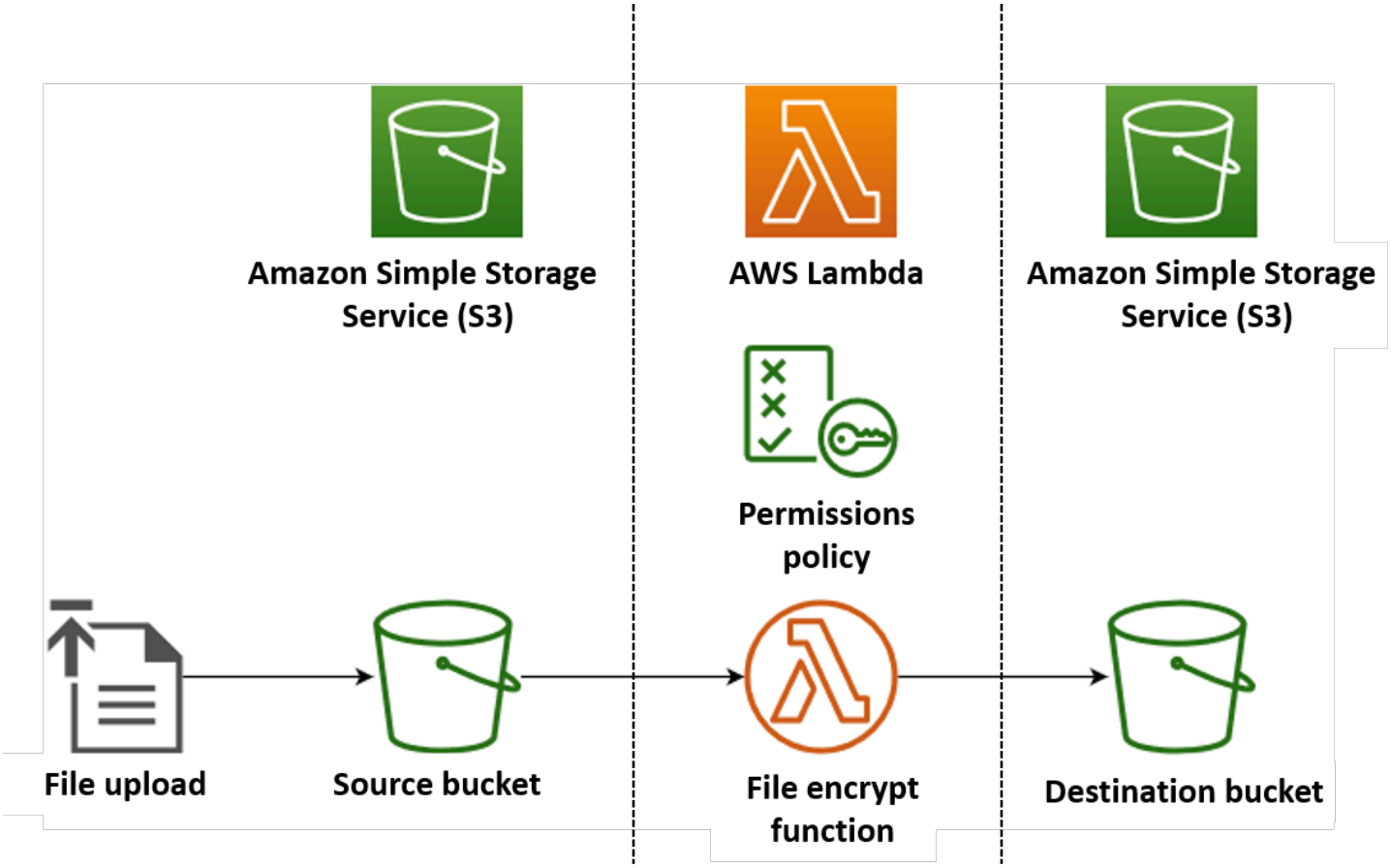
Erstellen Sie eine Serverless-Dateiverarbeitungs-App

Einer der häufigsten Anwendungsfälle für Lambda ist die Ausführung von Dateiverarbeitungsaufgaben. Sie können beispielsweise eine Lambda-Funktion verwenden, um automatisch PDF-Dateien aus HTML-Dateien oder Bildern zu erstellen oder um Miniaturansichten zu erstellen, wenn ein Benutzer ein Bild hochlädt.

In diesem Beispiel erstellen Sie eine App, die PDF-Dateien automatisch verschlüsselt, wenn sie in einen Amazon Simple Storage Service (Amazon S3)-Bucket uploaden. Um diese Anwendung zu implementieren, erstellen Sie die folgenden Ressourcen:

- Ein S3-Bucket, in das Benutzer PDF-Dateien hochladen können
- Eine Lambda-Funktion in Python, die die hochgeladene Datei liest und eine verschlüsselte, passwortgeschützte Version davon erstellt
- Ein zweiter S3-Bucket für Lambda zum Speichern der verschlüsselten Datei

Sie erstellen auch eine AWS Identity and Access Management (IAM-) Richtlinie, um Ihrer Lambda-Funktion die Erlaubnis zu erteilen, Lese- und Schreibvorgänge auf Ihren S3-Buckets durchzuführen.



> **Tip**
>
> Wenn Sie Lambda noch nicht kennen, empfehlen wir Ihnen, mit dem Tutorial zu beginnen, the section called “Erstellen Sie Ihre erste Funktion” bevor Sie diese Beispiel-App erstellen.

Sie können Ihre App manuell bereitstellen, indem Sie Ressourcen mit dem AWS-Managementkonsole oder dem AWS Command Line Interface (AWS CLI) erstellen und konfigurieren. Sie können die App auch mithilfe von AWS Serverless Application Model (AWS SAM) bereitstellen. AWS SAM ist ein Tool für Infrastruktur als Code (IaC). Mit IaC erstellen Sie Ressourcen nicht manuell, sondern definieren sie im Code und stellen sie dann automatisch bereit.

Wenn Sie mehr über die Verwendung von Lambda mit IaC erfahren möchten, bevor Sie diese Beispiel-App bereitstellen, finden Sie unter the section called “Infrastructure as Code (IaC)”.

Erstellen der Quellcodedateien für die Lambda-Funktion

Erstellen Sie die folgenden Dateien in Ihrem Projektverzeichnis:

- `lambda_function.py` - Python-Funktionscode für die Lambda-Funktion, die die Dateiverschlüsselung durchführt
- `requirements.txt` - eine Manifestdatei, die die Abhängigkeiten definiert, die Ihr Python-Funktionscode benötigt

Erweitern Sie die folgenden Abschnitte, um den Code anzusehen und mehr über die Rolle der einzelnen Dateien zu erfahren. Um die Dateien auf Ihrem lokalen Computer zu erstellen, kopieren Sie entweder den folgenden Code und fügen Sie ihn ein, oder laden Sie die Dateien aus dem GitHub aws-lambda-developer-guide-Repository herunter.

Python-Funktionscode

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `lambda_function.py` ein.

```
from pypdf import PdfReader, PdfWriter
import uuid
import os
from urllib.parse import unquote_plus
import boto3

# Create the S3 client to download and upload objects from S3
s3_client = boto3.client('s3')

def lambda_handler(event, context):
    # Iterate over the S3 event object and get the key for all uploaded files
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key']) # Decode the S3 object key to
 remove any URL-encoded characters
        download_path = f'/tmp/{uuid.uuid4()}.pdf' # Create a path in the Lambda tmp
 directory to save the file to
        upload_path = f'/tmp/converted-{uuid.uuid4()}.pdf' # Create another path to
 save the encrypted file to

        # If the file is a PDF, encrypt it and upload it to the destination S3 bucket
        if key.lower().endswith('.pdf'):
            s3_client.download_file(bucket, key, download_path)
            encrypt_pdf(download_path, upload_path)
            encrypted_key = add_encrypted_suffix(key)
            s3_client.upload_file(upload_path, f'{bucket}-encrypted', encrypted_key)
```

```
# Define the function to encrypt the PDF file with a password
def encrypt_pdf(file_path, encrypted_file_path):
    reader = PdfReader(file_path)
    writer = PdfWriter()

    for page in reader.pages:
        writer.add_page(page)

    # Add a password to the new PDF
    writer.encrypt("my-secret-password")

    # Save the new PDF to a file
    with open(encrypted_file_path, "wb") as file:
        writer.write(file)

# Define a function to add a suffix to the original filename after encryption
def add_encrypted_suffix(original_key):
    filename, extension = original_key.rsplit('.', 1)
    return f'{filename}_encrypted.{extension}'
```

> **Note**
>
> In diesem Beispielcode ist ein Passwort für die verschlüsselte Datei (`my-secret-password`) fest im Funktionscode codiert. Schließen Sie in einer Produktionsanwendung keine sensiblen Informationen wie Passwörter in Ihren Funktionscode ein. Erstellen Sie stattdessen ein AWS Secrets Manager Geheimnis und verwenden Sie dann die Lambda-Erweiterung AWS Parameters and Secrets, um Ihre Anmeldeinformationen in Ihrer Lambda-Ereignisfunktion abzurufen.

Der Python-Funktionscode enthält drei Funktionen: die Handler-Funktion, die Lambda ausführt, wenn Ihre Funktion aufgerufen wird und zwei separate Funktionen mit dem Namen `add_encrypted_suffix` und `encrypt_pdf`, die der Handler aufruft, um die PDF-Verschlüsselung durchzuführen.

Wenn Ihre Funktion von Amazon S3 aufgerufen wird, übergibt Lambda ein JSON-formatiertes Ereignisargument an die Funktion, das Details zu dem Ereignis enthält, das den Aufruf verursacht hat. In diesem Fall umfassen die Informationen den Namen des S3-Buckets und die Objektschlüssel

für die hochgeladenen Dateien. Weitere Informationen zum Format des Ereignisobjekts für Amazon S3 finden Sie unter the section called “S3”.

Ihre Funktion verwendet dann die AWS SDK für Python (Boto3) , um die im Ereignisobjekt angegebenen PDF-Dateien in ihr lokales temporäres Speicherverzeichnis herunterzuladen, bevor sie sie mit der pypdf-Bibliothek verschlüsselt.

Schließlich verwendet die Funktion das Boto3-SDK, um die verschlüsselte Datei in Ihrem S3-Ziel-Bucket zu speichern.

`Manifestdatei requirements.txt`

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `requirements.txt` ein.

```
boto3
pypdf
```

In diesem Beispiel hat Ihr Funktionscode nur zwei Abhängigkeiten, die nicht Teil der Standard-Python-Bibliothek sind: das SDK für Python (Boto3) und das `pypdf`-Paket, das die Funktion zur Durchführung der PDF-Verschlüsselung verwendet.

> **Note**
>
> Eine Version des SDK für Python (Boto3) ist Teil der Lambda-Laufzeit, sodass Ihr Code ausgeführt werden kann, ohne Boto3 zum Bereitstellungspaket Ihrer Funktion hinzuzufügen. Um jedoch die volle Kontrolle über die Abhängigkeiten Ihrer Funktion zu behalten und mögliche Probleme mit Versionsfehlern zu vermeiden, empfiehlt es sich für Python, alle Funktionsabhängigkeiten in das Bereitstellungspaket Ihrer Funktion aufzunehmen. Weitere Informationen hierzu finden Sie unter the section called “Laufzeitabhängigkeiten in Python”.

Bereitstellen der App

Sie können die Ressourcen für diese Beispiel-App entweder manuell oder mithilfe von erstellen und bereitstellen AWS SAM. In einer Produktionsumgebung empfehlen wir, ein IaC-Tool zu verwenden, AWS SAM um ganze serverlose Anwendungen schnell und wiederholbar bereitzustellen, ohne manuelle Prozesse zu verwenden.

Stellen Sie die Ressourcen manuell bereit

So stellen Sie Ihre App manuell bereit:

- Erstellen Sie Quell- und Ziel-Amazon-S3-Buckets
- Erstellen Sie eine Lambda-Funktion, die eine PDF-Datei verschlüsselt und die verschlüsselte Version in einem S3-Bucket speichert
- Konfigurieren Sie einen Lambda-Auslöser, der Ihre Funktion aufruft, wenn Objekte in Ihren Quell-Bucket hochgeladen werden

Stellen Sie zunächst sicher, dass Python auf Ihrer Build-Maschine installiert ist.

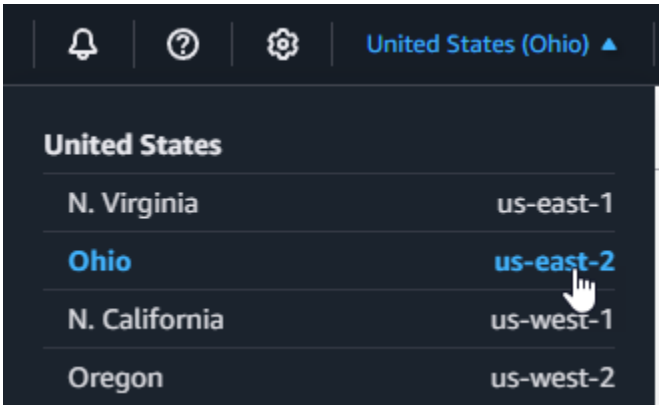
Erstellen Sie zwei S3-Buckets

Erstellen Sie zuerst zwei S3-Buckets. Der erste Bucket ist der Quell-Bucket, in den Sie Ihre PDF-Dateien hochladen. Der zweite Bucket wird von Lambda zum Speichern der verschlüsselten Datei verwendet, wenn Sie Ihre Funktion aufrufen.

Console

So erstellen Sie die S3-Buckets (Konsole)

1. Öffnen Sie die Seite Allzweck-Buckets in der Amazon-S3-Konsole.
2. Wählen Sie den, der Ihrem geografischen Standort AWS-Region am nächsten liegt. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern.



3. Wählen Sie Create Bucket (Bucket erstellen) aus.
4. Führen Sie unter Allgemeine Konfiguration die folgenden Schritte aus:

   a. Stellen Sie sicher, dass für Bucket-Typ die Option Allzweck ausgewählt ist.

b. Geben Sie für den Bucket-Namen einen global eindeutigen Namen ein, der den Regeln für die Bucket-Benennung von Amazon S3 entspricht. Bucket-Namen dürfen nur aus Kleinbuchstaben, Zahlen, Punkten (.) und Bindestrichen (-) bestehen.

5. Belassen Sie alle anderen Optionen auf ihren Standardwerten und wählen Sie Bucket erstellen aus.
6. Wiederholen Sie die Schritte 1 bis 4, um Ihren Ziel-Bucket zu erstellen. Geben Sie unter Bucket-Name den Namen **`amzn-s3-demo-bucket-encrypted`** ein. **`amzn-s3-demo-bucket`** ist hierbei der Name des Quell-Buckets, den Sie soeben erstellt haben.

AWS CLI

Stellen Sie zunächst sicher, dass die AWS CLI auf Ihrer Build-Maschine installiert ist.

Um die Amazon S3 S3-Buckets zu erstellen (AWS CLI)

1. Führen Sie den folgenden CLI-Befehl aus, um Ihren Quell-Bucket zu erstellen. Der gewählte Name für Ihren Bucket muss global eindeutig sein und den Regeln für die Benennung von Buckets von Amazon S3 entsprechen. Namen dürfen nur Kleinbuchstaben, Zahlen, Punkte (.) und Bindestriche (-) enthalten. Wählen Sie für `region` und `LocationConstraint` die AWS-Region aus, die Ihrem geografischen Standort am nächsten ist.

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket --region us-east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

   Später im Tutorial müssen Sie Ihre Lambda-Funktion im selben Verzeichnis AWS-Region wie in Ihrem Quell-Bucket erstellen. Notieren Sie sich also die Region, die Sie ausgewählt haben.

2. Führen Sie den folgenden Befehl aus, um Ihren Ziel-Bucket zu erstellen. Verwenden Sie für den Bucket-Namen **`amzn-s3-demo-bucket-encrypted`**. **`amzn-s3-demo-bucket`** ist hierbei der Name des Quell-Buckets, den Sie in Schritt 1 erstellt haben. Wählen Sie für `region` und dasselbe aus`LocationConstraint`, das AWS-Region Sie zur Erstellung Ihres Quell-Buckets verwendet haben.

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket-encrypted --region us-
east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

Erstellen einer Ausführungsrolle

Eine Ausführungsrolle ist eine IAM-Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. Fügen Sie die AWS -verwaltete Richtlinie `AmazonS3FullAccess` an, um Ihrer Funktion Lese- und Schreibzugriff auf Amazon S3 zu erteilen.

Console

So erstellen Sie eine Ausführungsrolle und fügen die verwaltete **`AmazonS3FullAccess`**-Richtlinie an (Konsole)

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS -Service und dann für Anwendungsfall die Option Lambda aus.
4. Wählen Sie Weiter aus.
5. Fügen Sie die verwaltete `AmazonS3FullAccess`-Richtlinie wie folgt hinzu:
   a. Geben Sie unter Berechtigungsrichtlinien die Zeichenfolge **`AmazonS3FullAccess`** in das Suchfeld ein.
   b. Aktivieren Sie das Kontrollkästchen neben der Richtlinie.
   c. Wählen Sie Weiter aus.
6. Geben Sie unter Rollendetails für Rollenname den Namen **`LambdaS3Role`** ein.
7. Wählen Sie Rolle erstellen aus.

AWS CLI

Um eine Ausführungsrolle zu erstellen und die **`AmazonS3FullAccess`**verwaltete Richtlinie anzuhängen (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `trust-policy.json`: Diese Vertrauensrichtlinie ermöglicht es Lambda, die Berechtigungen der Rolle zu verwenden, indem dem Dienstprinzipal die `lambda.amazonaws.com` Erlaubnis erteilt wird, die `AssumeRole` Aktion AWS -Security-Token-Service (AWS STS) aufzurufen.

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. Führen Sie in dem Verzeichnis, in dem Sie das JSON-Vertrauensrichtliniendokument gespeichert haben, den folgenden CLI-Befehl aus, um die Ausführungsrolle zu erstellen:

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. Führen Sie den folgenden CLI-Befehl aus, um die `AmazonS3FullAccess`-verwaltete Richtlinie anzuhängen.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::aws:policy/AmazonS3FullAccess
```

Erstellen des Bereitstellungspakets für die Funktion

Um Ihre Funktion zu erstellen, erstellen Sie ein Bereitstellungspaket, das Ihren Funktionscode und die zugehörigen Abhängigkeiten enthält. Bei der hier verwendeten Anwendung verwendet Ihr Funktionscode eine separate Bibliothek für die PDF-Verschlüsselung.

Erstellen des Bereitstellungspakets

1. Navigieren Sie zu dem Projektverzeichnis, das die `lambda_function.py` `requirements.txt` Dateien enthält, die Sie GitHub zuvor erstellt oder heruntergeladen haben, und erstellen Sie ein neues Verzeichnis mit dem Namen`package`.
2. Installieren Sie die in der `requirements.txt`-Datei angegebenen Abhängigkeiten in Ihrem `package`-Verzeichnis, indem Sie den folgenden Befehl ausführen.

```
pip install -r requirements.txt --target ./package/
```

3. Erstellen Sie eine .zip-Datei, die Ihren Anwendungscode und seine Abhängigkeiten enthält. Führen Sie unter Linux oder MacOS die folgenden Befehle über Ihre Befehlszeilenschnittstelle aus:

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

   Verwenden Sie unter Windows Ihr bevorzugtes ZIP-Tool, um die `lambda_function.zip`-Datei zu erstellen. Achten Sie darauf, dass sich die Datei `lambda_function.py` und die Ordner, die Ihre Abhängigkeiten enthalten, im Stammverzeichnis der ZIP-Datei befinden.

Sie können Ihr Bereitstellungspaket auch in einer virtuellen Python-Umgebung erstellen. Siehe Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions

So erstellen Sie die Lambda-Funktion:

Sie verwenden jetzt das Bereitstellungspaket, das Sie im vorherigen Schritt erstellt haben, um Ihre Lambda-Funktion bereitzustellen.

Console

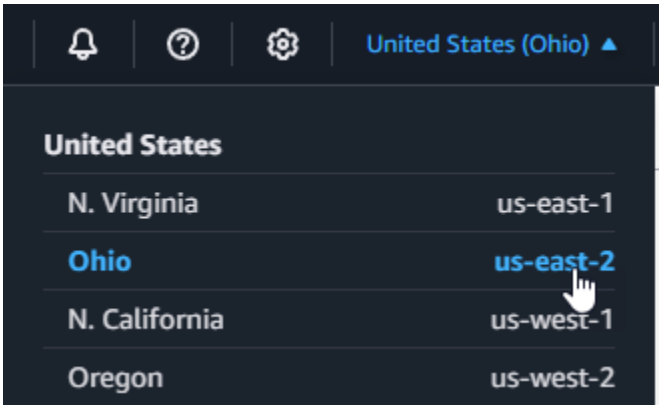
So erstellen Sie die Funktion (Konsole)

Um Ihre Lambda-Funktion mit der Konsole zu erstellen, erstellen Sie zunächst eine Basisfunktion, die etwas „Hello World“ -Code enthält. Anschließend ersetzen Sie diesen Code durch Ihren eigenen Funktionscode, indem Sie die im vorherigen Schritt erstellte ZIP- oder Datei hochladen.

Um sicherzustellen, dass Ihre Funktion beim Verschlüsseln großer PDF-Dateien nicht zu einem Timeout kommt, konfigurieren Sie die Speicher- und Timeout-Einstellungen der Funktion. Sie legen auch das Protokollformat der Funktion auf JSON fest. Die Konfiguration von Protokollen im JSON-Format ist erforderlich, wenn Sie das bereitgestellte Testskript verwenden, damit es den Aufrufstatus der Funktion aus den CloudWatch Protokollen lesen kann, um den erfolgreichen Aufruf zu bestätigen.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Stellen Sie sicher, dass Sie in demselben Modus arbeiten, in dem AWS-Region Sie Ihren S3-Bucket erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern.



3. Wählen Sie „Event-Funktion erstellen“.
4. Wählen Sie Author from scratch aus.
5. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Function name (Funktionsname) **`EncryptPDF`** ein.
   b. Wählen Sie für Laufzeit die Option Python 3.12 aus.
   c. Wählen Sie für Architektur x86_64 aus.
6. Fügen Sie die Ausführungsrolle, die Sie im vorherigen Schritt erstellt haben, wie folgt hinzu:
   a. Erweitern Sie den Abschnitt Ändern der standardmäßigen Ausführungsrolle.
   b. Wählen Sie Eine vorhandene Rolle verwenden aus.
   c. Wählen Sie unter Bestehende Rolle Ihre Rolle (`LambdaS3Role`) aus.
7. Wählen Sie „Event-Funktion erstellen“.

So laden Sie den Funktionscode hoch (Konsole)

1. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
2. Wählen Sie die ZIP-Datei aus.
3. Klicken Sie auf Upload.
4. Wählen Sie in der Dateiauswahl Ihre ZIP-Datei und anschließend Öffnen aus.
5. Wählen Sie Speichern.

Um den Funktionsspeicher und das Timeout (Konsole) zu konfigurieren

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie im Bereich Allgemeine Konfiguration die Option Bearbeiten aus.
3. Stellen Sie Speicher auf 256 MB und Timeout auf 15 Sekunden ein.
4. Wählen Sie Speichern.

So konfigurieren Sie das Protokollformat (Konsole)

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie Überwachungs- und Betriebstools aus.
3. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
4. Wählen Sie für die Konfiguration der Protokollierung JSON aus.
5. Wählen Sie Speichern.

AWS CLI

Um die Funktion zu erstellen (AWS CLI)

- Führen Sie den folgenden Befehl in dem Verzeichnis aus, das Ihre `lambda_function.zip`-Datei enthält. Ersetzen Sie `region` für den Parameter `us-east-2` durch die Region, in der Sie Ihre S3-Buckets erstellt haben.

```
aws lambda create-function --function-name EncryptPDF \
--zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler
 \
--runtime python3.12 --timeout 15 --memory-size 256 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-2 \
--logging-config LogFormat=JSON
```

Konfigurieren von Amazon-S3-Auslöser zum Aufrufen der Funktion

Damit Ihre Lambda-Funktion ausgeführt wird, wenn Sie eine Datei in Ihren Quell-Bucket hochladen, müssen Sie einen Auslöser für Ihre Funktion konfigurieren. Sie können den Amazon-S3-Auslöser entweder über die Konsole oder mithilfe der AWS CLI konfigurieren.

> **Important**
>
> Mit diesem Verfahren wird der S3-Bucket so konfiguriert, dass die Funktion jedes Mal aufgerufen wird, wenn ein Objekt im Bucket erstellt wird. Konfigurieren Sie dieses Verhalten nur für den Quell-Bucket. Wenn Ihre Lambda-Funktion Objekte in dem Bucket erstellt, der sie aufruft, kann es passieren, dass Ihre Funktion kontinuierlich in einer Schleife aufgerufen wird. Dies kann dazu führen, dass Ihnen AWS-Konto unerwartete Gebühren in Rechnung gestellt werden.

Console

So konfigurieren Sie den Amazon-S3-Auslöser (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion (`EncryptPDF`) aus.
2. Wählen Sie Add trigger.
3. Wählen Sie S3 aus.
4. Wählen Sie unter Bucket Ihren Quell-Bucket aus.
5. Wählen Sie unter Ereignistypen die Option Alle Objekterstellungsereignisse aus.
6. Aktivieren Sie unter Rekursiver Aufruf das Kontrollkästchen, um zu bestätigen, dass die Verwendung desselben S3-Buckets für die Ein- und Ausgabe nicht empfohlen wird. Weitere Informationen zu rekursiven Aufrufmustern in Lambda finden Sie bei Serverless Land unter Recursive patterns that cause run-away Lambda functions.
7. Wählen Sie Hinzufügen aus.

   Wenn Sie einen Auslöser über die Lambda-Konsole erstellen, erstellt Lambda automatisch eine ressourcenbasierte Richtlinie, um dem von Ihnen ausgewählten Dienst die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen.

AWS CLI

Um den Amazon S3-Trigger zu konfigurieren (AWS CLI)

1. Fügen Sie Ihrer Funktion eine ressourcenbasierte Richtlinie hinzu, die es Ihrem Amazon-S3-Quell-Bucket ermöglicht, Ihre Funktion aufzurufen, wenn Sie eine Datei hinzufügen. Eine ressourcenbasierte Richtlinienerklärung erteilt anderen die AWS-Services Erlaubnis,

Ihre Funktion aufzurufen. Führen Sie den folgenden CLI-Befehl aus, um Amazon S3 die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen. Achten Sie darauf, den `source-account` Parameter durch Ihre eigene AWS-Konto ID zu ersetzen und Ihren eigenen Quell-Bucket-Namen zu verwenden.

```
aws lambda add-permission --function-name EncryptPDF \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-bucket \
--source-account 123456789012
```

Die mit diesem Befehl definierte Richtlinie ermöglicht es Amazon S3 nur, Ihre Funktion aufzurufen, wenn eine Aktion in Ihrem Quell-Bucket ausgeführt wird.

> **Note**
>
> Namen von S3-Buckets sind zwar global eindeutig, trotzdem empfiehlt es sich, bei der Verwendung ressourcenbasierter Richtlinien anzugeben, dass der Bucket zu Ihrem Konto gehören muss. Denn wenn Sie einen Bucket löschen, kann ein anderer AWS-Konto Bucket einen Bucket mit demselben Amazon-Ressourcennamen (ARN) erstellen.

2. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `notification.json`: Wenn dieser JSON-Code auf Ihren Quell-Bucket angewendet wird, wird der Bucket so konfiguriert, dass er jedes Mal, wenn ein neues Objekt hinzugefügt wird, eine Benachrichtigung an Ihre Lambda-Funktion sendet. Ersetzen Sie die AWS-Konto Nummer und AWS-Region in der Lambda-Funktion ARN durch Ihre eigene Kontonummer und Region.

```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "EncryptPDFEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-2:123456789012:function:EncryptPDF",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Führen Sie den folgenden CLI-Befehl aus, um die Benachrichtigungseinstellungen in der von Ihnen erstellten JSON-Datei auf Ihren Quell-Bucket anzuwenden. Ersetzen Sie dabei `amzn-s3-demo-bucket` durch den Namen Ihres Quell-Buckets.

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-bucket \
--notification-configuration file://notification.json
```

Weitere Informationen zu dem `put-bucket-notification-configuration` Befehl und der `notification-configuration` Option finden Sie unter put-bucket-notification-configuration in der AWS CLI-Befehlsreferenz.

Stellen Sie die Ressourcen bereit mit AWS SAM

Stellen Sie zunächst sicher, dass Docker und die neueste Version von AWS SAM CLI auf Ihrer Build-Maschine installiert sind.

1. Kopieren Sie im Projektverzeichnis den folgenden Code und fügen Sie ihn in eine Datei namens `template.yaml` ein. Ersetzen Sie die Platzhalter-Bucket-Namen:

   - Ersetzen Sie für den Quell-Bucket `amzn-s3-demo-bucket` durch einen beliebigen Namen, der den Benennungsregeln für S3-Buckets entspricht.
   - Ersetzen Sie für den Zielbereich `amzn-s3-demo-bucket-encrypted` durch `<source-bucket-name>-encrypted`, wobei `<source-bucket>` der Name ist, den Sie für Ihren Quellbereich gewählt haben.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  EncryptPDFFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: EncryptPDF
      Architectures: [x86_64]
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.12
      Timeout: 15
      MemorySize: 256
```

```
      LoggingConfig:
        LogFormat: JSON
      Policies:
        - AmazonS3FullAccess
      Events:
        S3Event:
          Type: S3
          Properties:
            Bucket: !Ref PDFSourceBucket
            Events: s3:ObjectCreated:*

  PDFSourceBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket

  EncryptedPDFBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket-encrypted
```

Die AWS SAM Vorlage definiert die Ressourcen, die Sie für Ihre App erstellen. In diesem Beispiel definiert die Vorlage eine Lambda-Funktion unter Verwendung des `AWS::Serverless::Function`-Typs und zwei S3-Buckets unter Verwendung des `AWS::S3::Bucket`-Typs. Die in der Vorlage angegebenen Bucket-Namen sind Platzhalter. Bevor Sie die App mithilfe bereitstellen AWS SAM, müssen Sie die Vorlage bearbeiten, um die Buckets mit global eindeutigen Namen umzubenennen, die den S3-Bucket-Namensregeln entsprechen. Dieser Schritt wird unter the section called “Stellen Sie die Ressourcen bereit mit AWS SAM” näher erläutert.

Die Definition der Lambda-Funktionsressource konfiguriert einen Trigger für die Funktion mithilfe der `S3Event`-Event-Eigenschaft. Dieser Trigger bewirkt, dass Ihre Funktion immer dann aufgerufen wird, wenn ein Objekt in Ihrem Quell-Bucket erstellt wird.

Die Funktionsdefinition spezifiziert auch eine AWS Identity and Access Management (IAM-) Richtlinie, die der Ausführungsrolle der Funktion zugeordnet werden soll. Die AWS -verwaltete Richtlinie `AmazonS3FullAccess` gibt Ihrer Funktion die Berechtigungen, die sie zum Lesen und Schreiben von Objekten in Amazon S3 benötigt.

2. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihre `template.yaml`-, `lambda_function.py`- und `requirements.txt`-Dateien gespeichert haben.

```
sam build --use-container
```

Dieser Befehl sammelt die Build-Artefakte für Ihre Anwendung und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen. Wenn Sie die `--use-container` Option angeben, wird Ihre Funktion in einem Lambda-like Docker-Container erstellt. Wir verwenden es hier, sodass Sie Python 3.12 nicht auf Ihrem lokalen Computer installiert haben müssen, damit der Build funktioniert.

Sucht während des Erstellungsvorgangs nach dem Lambda-Funktionscode an der Stelle, die Sie mit der `CodeUri` Eigenschaft in der Vorlage angegeben haben. AWS SAM In diesem Fall haben wir das aktuelle Verzeichnis als Standort (`./`) angegeben.

Wenn eine `requirements.txt` Datei vorhanden ist, AWS SAM verwendet sie, um die angegebenen Abhängigkeiten zu sammeln. AWS SAM Erstellt standardmäßig ein ZIP-Bereitstellungspaket mit Ihrem Funktionscode und Ihren Abhängigkeiten. Mithilfe der PackageTypeEigenschaft können Sie Ihre Funktion auch als Container-Image bereitstellen.

3. Führen Sie den folgenden Befehl aus, um Ihre Anwendung bereitzustellen und die in Ihrer AWS SAM Vorlage angegebenen Lambda- und Amazon S3 S3-Ressourcen zu erstellen.

```
sam deploy --guided
```

Wenn Sie die `--guided` Markierung verwenden, AWS SAM werden Ihnen Eingabeaufforderungen angezeigt, die Sie durch den Bereitstellungsprozess führen. Akzeptieren Sie für diese Bereitstellung die Standardoptionen, indem Sie die Eingabetaste drücken.

AWS SAM Erstellt während des Bereitstellungsprozesses die folgenden Ressourcen in Ihrem AWS-Konto:

- Ein CloudFormation Stack mit dem Namen `sam-app`
- Eine Lambda-Funktion mit dem Namen `EncryptPDF`
- Zwei S3-Buckets mit den Namen, die Sie bei der Bearbeitung der `template.yaml` AWS SAM Vorlagendatei ausgewählt haben
- Eine IAM-Ausführungsrolle für Ihre Funktion mit dem Namensformat `sam-app-EncryptPDFFunctionRole-2qGaapHFWOQ8`

Wenn Sie mit der Erstellung Ihrer Ressourcen AWS SAM fertig sind, sollte die folgende Meldung angezeigt werden:

```
Successfully created/updated stack - sam-app in us-east-2
```

Testen der Anwendung

Um Ihre App zu testen, laden Sie eine PDF-Datei in Ihren Quell-Bucket hoch und bestätigen, dass Lambda eine verschlüsselte Version der Datei in Ihrem Ziel-Bucket erstellt. In diesem Beispiel können Sie dies entweder manuell mit der Konsole oder dem AWS CLI oder mithilfe des bereitgestellten Testskripts testen.

Für Produktionsanwendungen können Sie herkömmliche Testmethoden und -techniken wie Modultests verwenden, um die korrekte Funktionsweise Ihres Lambda-Funktionscodes zu überprüfen. Es empfiehlt sich auch, Tests wie die im bereitgestellten Testskript enthaltenen durchzuführen, bei denen Integrationstests mit echten, cloudbasierten Ressourcen durchgeführt werden. Integrationstests in der Cloud bestätigen, dass Ihre Infrastruktur korrekt bereitgestellt wurde und dass Ereignisse erwartungsgemäß zwischen verschiedenen Diensten ablaufen. Weitere Informationen hierzu finden Sie unter the section called “Testen”.

Manuelles Testen der App

Sie können Ihre Funktion manuell testen, indem Sie Ihrem Amazon-S3-Quell-Bucket eine PDF-Datei hinzufügen. Wenn Sie Ihre Datei dem Quell-Bucket hinzufügen, sollte Ihre Lambda-Funktion automatisch aufgerufen werden und eine verschlüsselte Version der Datei in Ihrem Ziel-Bucket speichern.

Console

So testen Sie Ihre Anwendung, indem Sie eine Datei hochladen (Konsole)

1. Um eine PDF-Datei in Ihren S3-Bucket hochzuladen, gehen Sie wie folgt vor:
   a. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Quell-Bucket aus.
   b. Klicken Sie auf Upload.
   c. Wählen Sie Dateien hinzufügen und anschließend über die Dateiauswahl die PDF-Datei aus, die Sie hochladen möchten.
   d. Wählen Sie Öffnen und anschließend Hochladen aus.

2. Überprüfen Sie, ob Lambda eine verschlüsselte Version Ihrer PDF-Datei in Ihrem Ziel-Bucket gespeichert hat, indem Sie Folgendes tun:

   a. Navigieren Sie wieder zur Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Ziel-Bucket aus.

   b. Im Objektbereich sollten Sie nun eine Datei mit dem Namen `filename_encrypted.pdf`-Format sehen (wobei `filename.pdf` der Name der Datei ist, die Sie in Ihren Quell-Bucket hochgeladen haben). Um Ihre verschlüsselte PDF-Datei herunterzuladen, wählen Sie die Datei aus und wählen Sie dann Herunterladen aus.

   c. Bestätigen Sie, dass Sie die heruntergeladene Datei mit dem Passwort öffnen können, mit dem Ihre Lambda-Funktion sie geschützt hat (`my-secret-password`).

AWS CLI

Um Ihre App zu testen, indem Sie eine Datei hochladen (AWS CLI)

1. Führen Sie in dem Verzeichnis, das die hochzuladende PDF-Datei enthält, den folgenden CLI-Befehl aus. Ersetzen Sie dabei den Parameter `--bucket` durch den Namen Ihres Quell-Buckets. Verwenden Sie für die `--key`- und `--body`-Parameter den Dateinamen Ihres Testbilds.

```
aws s3api put-object --bucket amzn-s3-demo-bucket --key test.pdf --body ./
test.pdf
```

2. Vergewissern Sie sich, dass Ihre Funktion eine verschlüsselte Version Ihrer Datei erstellt und in Ihrem S3-Ziel-Bucket gespeichert hat. Führen Sie den folgenden CLI-Befehl aus und ersetzen Sie dabei `amzn-s3-demo-bucket-encrypted` durch den Namen Ihres eigenen Ziel-Buckets:

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-bucket-encrypted
```

   Wenn Ihre Funktion erfolgreich ausgeführt wird, erhalten Sie eine Ausgabe, die der folgenden ähnelt. Ihr Ziel-Bucket sollte eine Datei mit dem Namensformat *<your_test_file>*`_encrypted.pdf` enthalten, wobei `<your_test_file>` der Name der von Ihnen hochgeladenen Datei ist.

```
{
    "Contents": [
```

```
        {
            "Key": "test_encrypted.pdf",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        }
    ]
}
```

3. Führen Sie den folgenden CLI-Befehl aus, um die Datei herunterzuladen, die Lambda in Ihrem Ziel-Bucket gespeichert hat. Ersetzen Sie den `--bucket`-Parameter durch den Namen Ihres Ziel-Buckets. Verwenden Sie für den Parameter `--key` den Dateinamen *<your_test_file>*`_encrypted.pdf`, wobei <your_test_file> der Name der von Ihnen hochgeladenen Testdatei ist.

```
aws s3api get-object --bucket amzn-s3-demo-bucket-encrypted --
key test_encrypted.pdf my_encrypted_file.pdf
```

   Dieser Befehl lädt die Datei in Ihr aktuelles Verzeichnis herunter und speichert sie als `my_encrypted_file.pdf`.

4. Bestätigen Sie, dass Sie die heruntergeladene Datei mit dem Passwort öffnen können, mit dem Ihre Lambda-Funktion sie geschützt hat (`my-secret-password`).

Testen Sie die App mit dem automatisierten Skript

Erstellen Sie die folgenden Dateien in Ihrem Projektverzeichnis:

- `test_pdf_encrypt.py` - ein Testskript, mit dem Sie Ihre Anwendung automatisch testen können
- `pytest.ini` - eine Konfigurationsdatei für das Testskript

Erweitern Sie die folgenden Abschnitte, um den Code anzusehen und mehr über die Rolle der einzelnen Dateien zu erfahren.

Automatisierte Testskripte

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `test_pdf_encrypt.py` ein. Ersetzen Sie unbedingt die Platzhalter-Bucket-Namen:

- Ersetzen Sie in der `test_source_bucket_available`-Funktion `amzn-s3-demo-bucket` durch den Namen Ihres Quell-Eimers.
- Ersetzen Sie in der Funktion `test_encrypted_file_in_bucket` `amzn-s3-demo-bucket-encrypted` durch *`source-bucket`*`-encrypted`, wobei `source-bucket>` der Name Ihres Quell-Eimers ist.
- Ersetzen Sie in der `cleanup`-Funktion `amzn-s3-demo-bucket` durch den Namen Ihres Quell-Buckets und ersetzen Sie `amzn-s3-demo-bucket-encrypted` durch den Namen Ihres Ziel-Buckets.

```
import boto3
import json
import pytest
import time
import os

@pytest.fixture
def lambda_client():
    return boto3.client('lambda')

@pytest.fixture
def s3_client():
    return boto3.client('s3')

@pytest.fixture
def logs_client():
    return boto3.client('logs')

@pytest.fixture(scope='session')
def cleanup():
    # Create a new S3 client for cleanup
    s3_client = boto3.client('s3')

    yield
    # Cleanup code will be executed after all tests have finished

    # Delete test.pdf from the source bucket
    source_bucket = 'amzn-s3-demo-bucket'
    source_file_key = 'test.pdf'
    s3_client.delete_object(Bucket=source_bucket, Key=source_file_key)
    print(f"\nDeleted {source_file_key} from {source_bucket}")
```

```
    # Delete test_encrypted.pdf from the destination bucket
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    destination_file_key = 'test_encrypted.pdf'
    s3_client.delete_object(Bucket=destination_bucket, Key=destination_file_key)
    print(f"Deleted {destination_file_key} from {destination_bucket}")


@pytest.mark.order(1)
def test_source_bucket_available(s3_client):
    s3_bucket_name = 'amzn-s3-demo-bucket'
    file_name = 'test.pdf'
    file_path = os.path.join(os.path.dirname(__file__), file_name)

    file_uploaded = False
    try:
        s3_client.upload_file(file_path, s3_bucket_name, file_name)
        file_uploaded = True
    except:
        print("Error: couldn't upload file")

    assert file_uploaded, "Could not upload file to S3 bucket"



@pytest.mark.order(2)
def test_lambda_invoked(logs_client):

    # Wait for a few seconds to make sure the logs are available
    time.sleep(5)

    # Get the latest log stream for the specified log group
    log_streams = logs_client.describe_log_streams(
        logGroupName='/aws/lambda/EncryptPDF',
        orderBy='LastEventTime',
        descending=True,
        limit=1
    )

    latest_log_stream_name = log_streams['logStreams'][0]['logStreamName']

    # Retrieve the log events from the latest log stream
    log_events = logs_client.get_log_events(
        logGroupName='/aws/lambda/EncryptPDF',
        logStreamName=latest_log_stream_name
```

```
    )

    success_found = False
    for event in log_events['events']:
        message = json.loads(event['message'])
        status = message.get('record', {}).get('status')
        if status == 'success':
            success_found = True
            break

    assert success_found, "Lambda function execution did not report 'success' status in
 logs."

@pytest.mark.order(3)
def test_encrypted_file_in_bucket(s3_client):
    # Specify the destination S3 bucket and the expected converted file key
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    converted_file_key = 'test_encrypted.pdf'

    try:
        # Attempt to retrieve the metadata of the converted file from the destination
 S3 bucket
        s3_client.head_object(Bucket=destination_bucket, Key=converted_file_key)
    except s3_client.exceptions.ClientError as e:
        # If the file is not found, the test will fail
        pytest.fail(f"Converted file '{converted_file_key}' not found in the
 destination bucket: {str(e)}")

def test_cleanup(cleanup):
    # This test uses the cleanup fixture and will be executed last
    pass
```

Das automatisierte Testskript führt drei Testfunktionen aus, um den korrekten Betrieb Ihrer App zu bestätigen:

- Der Test `test_source_bucket_available` bestätigt, dass Ihr Quell-Bucket erfolgreich erstellt wurde, indem eine Test-PDF-Datei in den Bucket hochgeladen wird.
- Der Test `test_lambda_invoked` fragt den neuesten CloudWatch Log-Protokollstream für Ihre Funktion ab, um zu bestätigen, dass Ihre Lambda-Funktion beim Hochladen der Testdatei ausgeführt wurde und erfolgreich gemeldet wurde.
- Der Test `test_encrypted_file_in_bucket` bestätigt, dass Ihr Ziel-Bucket die verschlüsselte `test_encrypted.pdf`-Datei enthält.

Nachdem all diese Tests ausgeführt wurden, führt das Skript einen zusätzlichen Bereinigungsschritt aus, um die `test.pdf`- und `test_encrypted.pdf`-Dateien sowohl aus Ihrem Quell- als auch aus Ihrem Ziel-Bucket zu löschen.

Wie bei der AWS SAM Vorlage sind die in dieser Datei angegebenen Bucket-Namen Platzhalter. Bevor Sie den Test ausführen, müssen Sie diese Datei mit den echten Bucket-Namen Ihrer App bearbeiten. Dieser Schritt wird unter the section called “Testen Sie die App mit dem automatisierten Skript” näher erläutert

Testskript-Konfigurationsdatei

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `pytest.ini` ein.

```
[pytest]
markers =
    order: specify test execution order
```

Dies ist erforderlich, um die Reihenfolge anzugeben, in der die Tests im `test_pdf_encrypt.py`-Skript ausgeführt werden.

Um die Tests durchzuführen, gehen Sie wie folgt vor:

1. Stellen Sie sicher, dass das `pytest`-Modul in Ihrer lokalen Umgebung installiert ist. Sie können `pytest` installieren, indem Sie den folgenden Befehl ausführen:

   ```
   pip install pytest
   ```
2. Speichern Sie eine PDF-Datei mit dem Namen `test.pdf` in dem Verzeichnis, das die `test_pdf_encrypt.py`- und `pytest.ini`-Dateien enthält.
3. Öffnen Sie ein Terminal- oder Shell-Programm und führen Sie den folgenden Befehl in dem Verzeichnis aus, das die Testdateien enthält.

   ```
   pytest -s -v
   ```

   Wenn der Test abgeschlossen ist, sollten Sie eine Ausgabe wie die folgende sehen:

   ```
   ============================================================= test session starts
    ============================================================
   platform linux -- Python 3.12.2, pytest-7.2.2, pluggy-1.0.0 -- /usr/bin/python3
   ```

```
cachedir: .pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/
pdf_encrypt_app/.hypothesis/examples')
Test order randomisation NOT enabled. Enable with --random-order or --random-order-
bucket=<bucket_type>
rootdir: /home/pdf_encrypt_app, configfile: pytest.ini
plugins: anyio-3.7.1, hypothesis-6.70.0, localserver-0.7.1, random-order-1.1.0
collected 4 items

test_pdf_encrypt.py::test_source_bucket_available PASSED
test_pdf_encrypt.py::test_lambda_invoked PASSED
test_pdf_encrypt.py::test_encrypted_file_in_bucket PASSED
test_pdf_encrypt.py::test_cleanup PASSED
Deleted test.pdf from amzn-s3-demo-bucket
Deleted test_encrypted.pdf from amzn-s3-demo-bucket-encrypted


============================================================== 4 passed in 7.32s
 ===============================================================
```

Nächste Schritte

Nachdem Sie diese Beispiel-App erstellt haben, können Sie den bereitgestellten Code als Grundlage verwenden, um andere Arten von Dateiverarbeitungsanwendungen zu erstellen. Ändern Sie den Code in der `lambda_function.py`-Datei, um die Dateiverarbeitungslogik für Ihren Anwendungsfall zu implementieren.

Viele typische Anwendungsfälle für die Dateiverarbeitung beinhalten Bildverarbeitung. Bei der Verwendung von Python enthalten die beliebtesten Bildverarbeitungsbibliotheken wie Pillow typischerweise C- oder C++-Komponenten. Um sicherzustellen, dass das Bereitstellungspaket Ihrer Funktion mit der Lambda-Ausführungsumgebung kompatibel ist, ist es wichtig, die richtige Binärdatei für die Quelldistribution zu verwenden.

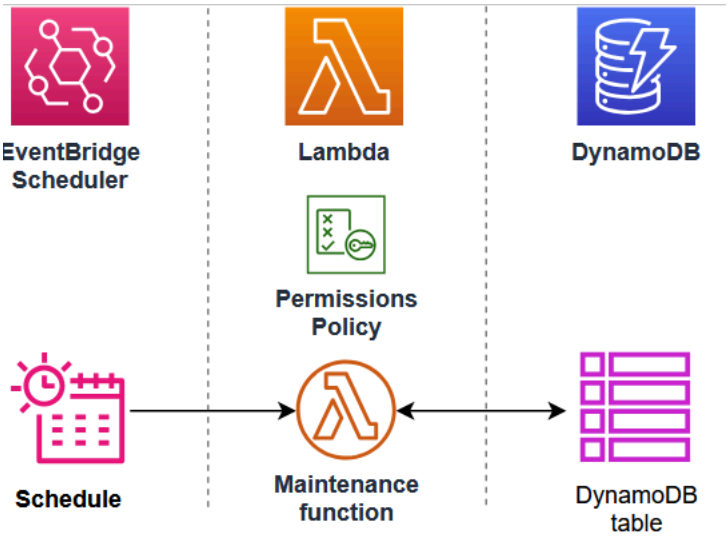
Bei der Bereitstellung Ihrer Ressourcen mit müssen Sie einige zusätzliche Schritte unternehmen AWS SAM, um die richtige Quelldistribution in Ihr Bereitstellungspaket aufzunehmen. Da AWS SAM keine Abhängigkeiten für eine andere Plattform als Ihre Build-Maschine installiert werden, funktioniert die Angabe der richtigen Quelldistribution (`.whl`Datei) in Ihrer `requirements.txt` Datei nicht, wenn Ihre Build-Maschine ein Betriebssystem oder eine Architektur verwendet, die sich von der Lambda-Ausführungsumgebung unterscheidet. Stattdessen sollten Sie eine der folgenden Möglichkeiten nutzen:

- Verwenden Sie die `--use-container`-Option beim Ausführen von `sam build`. Wenn Sie diese Option angeben, wird ein Container-Basis-Image AWS SAM heruntergeladen, das mit der Lambda-Ausführungsumgebung kompatibel ist, und erstellt das Bereitstellungspaket Ihrer Funktion mithilfe dieses Images in einem Docker-Container. Weitere Informationen finden Sie unter Eine Lambda-Funktion in einem bereitgestellten Container erstellen.
- Erstellen Sie das ZIP-Bereitstellungspaket Ihrer Funktion mithilfe der richtigen Binärdatei für die Quelldistribution selbst und speichern Sie die ZIP-Datei in dem Verzeichnis, das Sie in der `CodeUri` Vorlage angeben. AWS SAM Weitere Informationen zum Erstellen von .zip-Bereitstellungspaketen für Python mithilfe von Binärverteilungen finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen” und the section called “ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen”.

Erstellen Sie eine App, um eine geplante Datenbankwartung durchzuführen

Sie können AWS Lambda sie verwenden, um geplante Prozesse wie automatisierte Systemsicherungen, Dateikonvertierungen und Wartungsaufgaben zu ersetzen. In diesem Beispiel erstellen Sie eine Serverless-Anwendung, die regelmäßig geplante Wartungsarbeiten an einer DynamoDB-Tabelle durchführt, indem alte Einträge gelöscht werden. Die App verwendet EventBridge Scheduler, um eine Lambda-Funktion nach einem Cron-Zeitplan aufzurufen. Wenn die Funktion aufgerufen wird, fragt sie die Tabelle nach Elementen ab, die älter als ein Jahr sind und löscht sie. Die Funktion protokolliert jedes gelöschte Element in Logs. CloudWatch

Um dieses Beispiel zu implementieren, erstellen Sie zunächst eine DynamoDB-Tabelle und füllen Sie sie mit einigen Testdaten auf, damit Ihre Funktion sie abfragen kann. Erstellen Sie dann eine Python-Lambda-Funktion mit einem EventBridge Scheduler-Trigger und einer IAM-Ausführungsrolle, die der Funktion die Berechtigung erteilt, Elemente aus Ihrer Tabelle zu lesen und zu löschen.



> **Tip**
>
> Wenn Sie Lambda noch nicht kennen, empfehlen wir Ihnen, das Tutorial abzuschließen, the section called “Erstellen Sie Ihre erste Funktion” bevor Sie diese Beispiel-App erstellen.

Sie können Ihre App manuell bereitstellen, indem Sie Ressourcen mit dem AWS-Managementkonsole erstellen und konfigurieren. Sie können die App auch mithilfe von AWS Serverless Application Model (AWS SAM) bereitstellen. AWS SAM ist ein Tool für Infrastruktur als Code (IaC). Mit IaC erstellen Sie Ressourcen nicht manuell, sondern definieren sie im Code und stellen sie dann automatisch bereit.

Wenn Sie mehr über die Verwendung von Lambda mit IaC erfahren möchten, bevor Sie diese Beispiel-App bereitstellen, finden Sie unter the section called “Infrastructure as Code (IaC)”.

Voraussetzungen

Bevor Sie die Beispiel-App erstellen können, stellen Sie sicher, dass Sie die erforderlichen Befehlszeilentools und Programme installiert haben.

- Python

  Um die DynamoDB-Tabelle zu füllen, die Sie zum Testen Ihrer App erstellen, verwendet dieses Beispiel ein Python-Skript und eine CSV-Datei, um Daten in die Tabelle zu schreiben. Stellen Sie sicher, dass Python Version 3.8 oder höher auf Ihrem Computer installiert ist.
- AWS SAM CLI

Wenn Sie die DynamoDB-Tabelle erstellen und die Beispiel-App mithilfe bereitstellen möchten AWS SAM, müssen Sie die AWS SAM CLI installieren. Folgen Sie den Installationsanweisungen im AWS SAM Benutzerhandbuch.

- AWS CLI

  Um das bereitgestellte Python-Skript zum Auffüllen Ihrer Testtabelle zu verwenden, müssen Sie das AWS CLI installiert und konfiguriert haben. Das liegt daran, dass das Skript die verwendet AWS SDK für Python (Boto3), die Zugriff auf Ihre AWS Identity and Access Management (IAM-) Anmeldeinformationen benötigt. Sie benötigen außerdem die AWS CLI Installation, um Ressourcen mithilfe AWS SAM von bereitzustellen. Installieren Sie die Befehlszeilenschnittstelle, indem Sie die Installationsanweisungen im AWS Command Line Interface -Benutzerhandbuch befolgen.

- Docker

  Um die App mithilfe von bereitzustellen AWS SAM, muss Docker auch auf Ihrem Build-Computer installiert sein. Befolgen Sie die Anweisungen in Installieren der Docker-Engine auf der Docker-Dokumentations-Website.

Die Beispiel-App-Dateien werden heruntergeladen

Um die Beispieldatenbank und die Anwendung für die geplante Wartung zu erstellen, müssen Sie die folgenden Dateien in Ihrem Projektverzeichnis erstellen:

Beispieldatenbankdateien

- `template.yaml`- eine AWS SAM Vorlage, mit der Sie die DynamoDB-Tabelle erstellen können
- `sample_data.csv` - eine CSV-Datei mit Beispieldaten, die Sie in Ihre Tabelle laden können
- `load_sample_data.py` - ein Python-Skript, das die Daten in der CSV-Datei in die Tabelle schreibt

Scheduled-maintenance App-Dateien

- `lambda_function.py` - den Python-Funktionscode für die Lambda-Funktion, die die Datenbankwartung durchführt
- `requirements.txt` - eine Manifestdatei, die die Abhängigkeiten definiert, die Ihr Python-Funktionscode benötigt
- `template.yaml`- eine AWS SAM Vorlage, mit der Sie die App bereitstellen können

Datei testen

- `test_app.py` - ein Python-Skript, das die Tabelle scannt und den erfolgreichen Betrieb Ihrer Funktion bestätigt, indem es alle Datensätze ausgibt, die älter als ein Jahr sind

Erweitern Sie die folgenden Abschnitte, um den Code anzusehen und mehr über die Rolle der einzelnen Dateien beim Erstellen und Testen Ihrer App zu erfahren. Um die Dateien auf Ihrem lokalen Computer zu erstellen, kopieren Sie den folgenden Code und fügen Sie ihn ein.

AWS SAM Vorlage (Beispiel für eine DynamoDB-Tabelle)

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `template.yaml` ein.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for DynamoDB Table with Order_number as Partition Key and
 Date as Sort Key

Resources:
  MyDynamoDBTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: MyOrderTable
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: Order_number
          AttributeType: S
        - AttributeName: Date
          AttributeType: S
      KeySchema:
        - AttributeName: Order_number
          KeyType: HASH
        - AttributeName: Date
          KeyType: RANGE
      SSESpecification:
        SSEEnabled: true
      GlobalSecondaryIndexes:
        - IndexName: Date-index
          KeySchema:
```

```
            - AttributeName: Date
              KeyType: HASH
          Projection:
            ProjectionType: ALL
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true

Outputs:
  TableName:
    Description: DynamoDB Table Name
    Value: !Ref MyDynamoDBTable
  TableArn:
    Description: DynamoDB Table ARN
    Value: !GetAtt MyDynamoDBTable.Arn
```

> **Note**
>
> AWS SAM Vorlagen verwenden eine standardmäßige Benennungskonvention von. `template.yaml` In diesem Beispiel haben Sie zwei Vorlagendateien - eine zur Erstellung der Beispieldatenbank und eine weitere zur Erstellung der Anwendung selbst. Speichern Sie sie in separaten Unterverzeichnissen im Projektordner.

Diese AWS SAM Vorlage definiert die DynamoDB-Tabellenressource, die Sie zum Testen Ihrer App erstellen. Die Tabelle verwendet einen Primärschlüssel von `Order_number` mit einem Sortierschlüssel von `Date`. Damit Ihre Lambda-Funktion Elemente direkt nach Datum finden kann, definieren wir auch einen globalen Sekundärindex mit dem Namen `Date-index`.

Weitere Informationen zum Erstellen und Konfigurieren einer DynamoDB-Tabelle mithilfe der `AWS::DynamoDB::Table`-Ressource finden Sie unter AWS::DynamoDB::Table im AWS CloudFormation Benutzerhandbuch.

Beispiel für Datenbankdatendatei

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `sample_data.csv` ein.

```
Date,Order_number,CustomerName,ProductID,Quantity,TotalAmount
2023-09-01,ORD001,Alejandro Rosalez,PROD123,2,199.98
2023-09-01,ORD002,Akua Mansa,PROD456,1,49.99
2023-09-02,ORD003,Ana Carolina Silva,PROD789,3,149.97
```

```
2023-09-03,ORD004,Arnav Desai,PROD123,1,99.99
2023-10-01,ORD005,Carlos Salazar,PROD456,2,99.98
2023-10-02,ORD006,Diego Ramirez,PROD789,1,49.99
2023-10-03,ORD007,Efua Owusu,PROD123,4,399.96
2023-10-04,ORD008,John Stiles,PROD456,2,99.98
2023-10-05,ORD009,Jorge Souza,PROD789,3,149.97
2023-10-06,ORD010,Kwaku Mensah,PROD123,1,99.99
2023-11-01,ORD011,Li Juan,PROD456,5,249.95
2023-11-02,ORD012,Marcia Oliveria,PROD789,2,99.98
2023-11-03,ORD013,Maria Garcia,PROD123,3,299.97
2023-11-04,ORD014,Martha Rivera,PROD456,1,49.99
2023-11-05,ORD015,Mary Major,PROD789,4,199.96
2023-12-01,ORD016,Mateo Jackson,PROD123,2,199.99
2023-12-02,ORD017,Nikki Wolf,PROD456,3,149.97
2023-12-03,ORD018,Pat Candella,PROD789,1,49.99
2023-12-04,ORD019,Paulo Santos,PROD123,5,499.95
2023-12-05,ORD020,Richard Roe,PROD456,2,99.98
2024-01-01,ORD021,Saanvi Sarkar,PROD789,3,149.97
2024-01-02,ORD022,Shirley Rodriguez,PROD123,1,99.99
2024-01-03,ORD023,Sofia Martinez,PROD456,4,199.96
2024-01-04,ORD024,Terry Whitlock,PROD789,2,99.98
2024-01-05,ORD025,Wang Xiulan,PROD123,3,299.97
```

Diese Datei enthält einige Beispieltestdaten, mit denen Sie Ihre DynamoDB-Tabelle im CSV-Standardformat (durch Komma getrennte Werte) füllen können.

Python-Skript zum Laden von Beispieldaten

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `load_sample_data.py` ein.

```
import boto3
import csv
from decimal import Decimal

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MyOrderTable')
print("DDB client initialized.")

def load_data_from_csv(filename):
    with open(filename, 'r') as file:
        csv_reader = csv.DictReader(file)
```

```
        for row in csv_reader:
            item = {
                'Order_number': row['Order_number'],
                'Date': row['Date'],
                'CustomerName': row['CustomerName'],
                'ProductID': row['ProductID'],
                'Quantity': int(row['Quantity']),
                'TotalAmount': Decimal(str(row['TotalAmount']))
            }
            table.put_item(Item=item)
            print(f"Added item: {item['Order_number']} - {item['Date']}")

if __name__ == "__main__":
    load_data_from_csv('sample_data.csv')
    print("Data loading completed.")
```

Dieses Python-Skript verwendet zunächst die AWS SDK für Python (Boto3) , um eine Verbindung zu Ihrer DynamoDB-Tabelle herzustellen. Anschließend wird jede Zeile in der CSV-Datei mit den Beispieldaten durchlaufen, ein Element aus dieser Zeile erstellt und das Element mithilfe des boto3-SDK in die DynamoDB-Tabelle geschrieben.

Python-Funktionscode

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `lambda_function.py` ein.

```
import boto3
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Key, Attr
import logging

logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    # Initialize the DynamoDB client
    dynamodb = boto3.resource('dynamodb')

    # Specify the table name
    table_name = 'MyOrderTable'
    table = dynamodb.Table(table_name)

    # Get today's date
```

```
    today = datetime.now()

    # Calculate the date one year ago
    one_year_ago = (today - timedelta(days=365)).strftime('%Y-%m-%d')

    # Scan the table using a global secondary index
    response = table.scan(
        IndexName='Date-index',
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago
        }
    )

     # Delete old items
    with table.batch_writer() as batch:
        for item in response['Items']:
            Order_number = item['Order_number']
            batch.delete_item(
                Key={
                    'Order_number': Order_number,
                    'Date': item['Date']
                }
            )
            logger.info(f'deleted order number {Order_number}')

    # Check if there are more items to scan
    while 'LastEvaluatedKey' in response:
        response = table.scan(
            IndexName='DateIndex',
            FilterExpression='#date < :one_year_ago',
            ExpressionAttributeNames={
                '#date': 'Date'
            },
            ExpressionAttributeValues={
                ':one_year_ago': one_year_ago
            },
            ExclusiveStartKey=response['LastEvaluatedKey']
        )

        # Delete old items
```

```
        with table.batch_writer() as batch:
            for item in response['Items']:
                batch.delete_item(
                    Key={
                        'Order_number': item['Order_number'],
                        'Date': item['Date']
                    }
                )

    return {
        'statusCode': 200,
        'body': 'Cleanup completed successfully'
    }
```

Der Python-Funktionscode enthält die Handler-Funktion (`lambda_handler`), die Lambda ausführt, wenn Ihre Funktion aufgerufen wird.

Wenn die Funktion vom EventBridge Scheduler aufgerufen wird, verwendet sie die, um eine Verbindung AWS SDK für Python (Boto3) zu der DynamoDB-Tabelle herzustellen, für die die geplante Wartungsaufgabe ausgeführt werden soll. Anschließend wird mit Hilfe der Python `datetime`-Bibliothek das Datum vor einem Jahr berechnet, bevor die Tabelle nach Einträgen durchsucht wird, die älter als dieses Datum sind und diese gelöscht werden.

Beachten Sie, dass Antworten von DynamoDB-Abfrage- und Scanvorgängen auf eine maximale Größe von 1 MB begrenzt sind. Wenn die Antwort größer als 1 MB ist, paginiert DynamoDB die Daten und gibt ein `LastEvaluatedKey`-Element in der Antwort zurück. Um sicherzustellen, dass unsere Funktion alle Datensätze in der Tabelle verarbeitet, prüfen wir, ob dieser Schlüssel vorhanden ist und führen weiterhin Tabellenscans von der zuletzt ausgewerteten Position aus, bis die gesamte Tabelle gescannt wurde.

Manifestdatei requirements.txt

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `requirements.txt` ein.

```
boto3
```

In diesem Beispiel hat Ihr Funktionscode nur eine Abhängigkeit, die nicht Teil der Standard-Python-Bibliothek ist: das SDK für Python (Boto3), das die Funktion zum Scannen und Löschen von Elementen aus der DynamoDB-Tabelle verwendet.

> **Note**
>
> Eine Version des SDK für Python (Boto3) ist Teil der Lambda-Laufzeit, sodass Ihr Code ausgeführt werden kann, ohne Boto3 zum Bereitstellungspaket Ihrer Funktion hinzuzufügen. Um jedoch die volle Kontrolle über die Abhängigkeiten Ihrer Funktion zu behalten und mögliche Probleme mit Versionsfehlern zu vermeiden, empfiehlt es sich für Python, alle Funktionsabhängigkeiten in das Bereitstellungspaket Ihrer Funktion aufzunehmen. Weitere Informationen hierzu finden Sie unter the section called “Laufzeitabhängigkeiten in Python”.

AWS SAM Vorlage (App für geplante Wartung)

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `template.yaml` ein.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for Lambda function and EventBridge Scheduler rule

Resources:
  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: ScheduledDBMaintenance
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      Events:
        ScheduleEvent:
          Type: ScheduleV2
          Properties:
            ScheduleExpression: cron(0 3 1 * ? *)
            Description: Run on the first day of every month at 03:00 AM
      Policies:
        - CloudWatchLogsFullAccess
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Scan
                - dynamodb:BatchWriteItem
```

```
              Resource: !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/
MyOrderTable'

  LambdaLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub /aws/lambda/${MyLambdaFunction}
      RetentionInDays: 30

Outputs:
  LambdaFunctionName:
    Description: Lambda Function Name
    Value: !Ref MyLambdaFunction
  LambdaFunctionArn:
    Description: Lambda Function ARN
    Value: !GetAtt MyLambdaFunction.Arn
```

> **Note**
>
> AWS SAM Vorlagen verwenden eine Standardbenennungskonvention von. `template.yaml` In diesem Beispiel haben Sie zwei Vorlagendateien - eine zur Erstellung der Beispieldatenbank und eine weitere zur Erstellung der Anwendung selbst. Speichern Sie sie in separaten Unterverzeichnissen im Projektordner.

Diese AWS SAM Vorlage definiert die Ressourcen für Ihre App. Wir definieren die Lambda-Funktion mithilfe der `AWS::Serverless::Function`-Ressource. Der EventBridge Scheduler-Zeitplan und der Trigger zum Aufrufen der Lambda-Funktion werden mithilfe der `Events` Eigenschaft dieser Ressource unter Verwendung eines Typs von erstellt. `ScheduleV2` Weitere Informationen zum Definieren von EventBridge Scheduler-Zeitplänen in AWS SAM Vorlagen finden Sie unter ScheduleV2 im Entwicklerhandbuch.AWS Serverless Application Model

Zusätzlich zur Lambda-Funktion und dem EventBridge Scheduler-Zeitplan definieren wir auch eine CloudWatch Protokollgruppe für Ihre Funktion, an die Datensätze gelöschter Elemente gesendet werden sollen.

Skript testen

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `test_app.py` ein.

```
import boto3
```

```
from datetime import datetime, timedelta
import json

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')

# Specify your table name
table_name = 'YourTableName'
table = dynamodb.Table(table_name)

# Get the current date
current_date = datetime.now()

# Calculate the date one year ago
one_year_ago = current_date - timedelta(days=365)

# Convert the date to string format (assuming the date in DynamoDB is stored as a
 string)
one_year_ago_str = one_year_ago.strftime('%Y-%m-%d')

# Scan the table
response = table.scan(
    FilterExpression='#date < :one_year_ago',
    ExpressionAttributeNames={
        '#date': 'Date'
    },
    ExpressionAttributeValues={
        ':one_year_ago': one_year_ago_str
    }
)

# Process the results
old_records = response['Items']

# Continue scanning if we have more items (pagination)
while 'LastEvaluatedKey' in response:
    response = table.scan(
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago_str
        },
```

```
        ExclusiveStartKey=response['LastEvaluatedKey']
    )
    old_records.extend(response['Items'])

for record in old_records:
    print(json.dumps(record))

# The total number of old records should be zero.
print(f"Total number of old records: {len(old_records)}")
```

Dieses Testskript verwendet die AWS SDK für Python (Boto3) , um eine Verbindung zu Ihrer DynamoDB-Tabelle herzustellen und nach Elementen zu suchen, die älter als ein Jahr sind. Um zu überprüfen, ob die Lambda-Funktion erfolgreich ausgeführt wurde, druckt die Funktion am Ende des Tests die Anzahl der Datensätze, die älter als ein Jahr sind, noch in der Tabelle aus. Wenn die Lambda-Funktion erfolgreich war, sollte die Anzahl der alten Datensätze in der Tabelle Null sein.

DynamoDB-Beispieltabelle erstellen und auffüllen

Um Ihre App für geplante Wartungsarbeiten zu testen, erstellen Sie zunächst eine DynamoDB-Tabelle und füllen sie mit einigen Beispieldaten. Sie können die Tabelle entweder manuell mithilfe von AWS-Managementkonsole oder mithilfe von AWS SAM erstellen. Wir empfehlen Ihnen, die Tabelle mithilfe einiger AWS CLI Befehle schnell AWS SAM zu erstellen und zu konfigurieren.

Console

Erstellen einer DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie Create table (Tabelle erstellen) aus.
3. Erstellen Sie die Tabelle folgendermaßen:
   a. Geben Sie unter Tabellendetails für Tabellenname den Wert **`MyOrderTable`** ein.
   b. Geben Sie für Partitionsschlüssel **`Order_number`** ein und behalten Sie den Datentyp als Zeichenfolge bei.
   c. Geben Sie für Sortierschlüssel **`Date`** ein und behalten Sie den Datentyp als Zeichenfolge bei.
   d. Lassen Sie die Tabelleneinstellungen auf Standardeinstellungen eingestellt und wählen Sie Tabelle erstellen.

4. Wenn Ihre Tabelle mit der Erstellung fertig ist und ihr Status als Aktiv angezeigt wird, erstellen Sie einen globalen sekundären Index (GSI), indem Sie wie folgt vorgehen. Ihre App verwendet diesen globalen Index, um direkt anhand des Datums nach Elementen zu suchen, um zu bestimmen, welche gelöscht werden sollen.

   a. Wählen Sie MyOrderTableaus der Liste der Tabellen.

   b. Wählen Sie die Registerkarte Indizes.

   c. Wählen Sie unter Globale Sekundärindizes die Option Index erstellen aus.

   d. Geben Sie unter Indexdetails **`Date`** für den Partitionsschlüssel ein und lassen Sie den Datentyp auf String eingestellt.

   e. Geben Sie für Indexname **`Date-index`** ein.

   f. Verwenden Sie für alle anderen Parameter die Standardwerte, scrollen Sie ans Seitenende und wählen Sie Index erstellen aus.

AWS SAM

Erstellen einer DynamoDB-Tabelle

1. Navigieren Sie zu dem Ordner, in dem Sie die `template.yaml`-Datei für die DynamoDB-Tabelle gespeichert haben. Beachten Sie, dass in diesem Beispiel zwei `template.yaml`-Dateien verwendet werden. Vergewissern Sie sich, dass sie in separaten Unterordnern gespeichert sind und dass Sie sich im richtigen Ordner befinden, der die Vorlage zur Erstellung Ihrer DynamoDB-Tabelle enthält.
2. Führen Sie den folgenden Befehl aus.

   ```
   sam build
   ```

   Dieser Befehl sammelt die Build-Artefakte für die Ressourcen, die Sie bereitstellen möchten und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen.
3. Führen Sie den folgenden Befehl aus, um die in der `template.yaml`-Datei angegebene DynamoDB-Ressource zu erstellen.

   ```
   sam deploy --guided
   ```

   Wenn Sie die `--guided` Flagge verwenden, werden Ihnen Eingabeaufforderungen AWS SAM angezeigt, die Sie durch den Bereitstellungsprozess führen. Geben Sie für diese

Bereitstellung ein `Stack name` von **`cron-app-test-db`** ein und akzeptieren Sie die Standardeinstellungen für alle anderen Optionen mit der Eingabetaste.

Wenn AWS SAM die Erstellung der DynamoDB-Ressource abgeschlossen ist, sollte die folgende Meldung angezeigt werden.

```
Successfully created/updated stack - cron-app-test-db in us-west-2
```

4. Sie können außerdem überprüfen, ob die DynamoDB-Tabelle erstellt wurde, indem Sie die Tabellenseite der DynamoDB-Konsole öffnen. Sie sollten eine Tabelle mit dem Namen `MyOrderTable` sehen.

Nachdem Sie Ihre Tabelle erstellt haben, fügen Sie als Nächstes einige Beispieldaten hinzu, um Ihre App zu testen. Die CSV-Datei `sample_data.csv`, die Sie zuvor heruntergeladen haben, enthält eine Reihe von Beispieleinträgen, die aus Bestellnummern, Daten sowie Kunden- und Bestellinformationen bestehen. Verwenden Sie das bereitgestellte Python-Skript `load_sample_data.py`, um diese Daten zu Ihrer Tabelle hinzuzufügen.

So fügen Sie der Tabelle die Beispieldaten hinzu

1. Wechseln Sie in das Verzeichnis, das die `sample_data.csv`- und `load_sample_data.py`-Dateien enthält. Wenn sich diese Dateien in separaten Verzeichnissen befinden, verschieben Sie sie so, dass sie am selben Ort gespeichert werden.
2. Erstellen Sie eine virtuelle Python-Umgebung zur Ausführung des Skripts, indem Sie den folgenden Befehl ausführen. Wir empfehlen Ihnen, eine virtuelle Umgebung zu verwenden, da Sie im nächsten Schritt das AWS SDK für Python (Boto3) installieren müssen.

   ```
   python -m venv venv
   ```

3. Aktivieren Sie die virtuelle Umgebung, indem Sie den folgenden Befehl ausführen.

   ```
   source venv/bin/activate
   ```

4. Installieren Sie das SDK für Python (Boto3) in Ihrer virtuellen Umgebung, indem Sie den folgenden Befehl ausführen. Das Skript verwendet diese Bibliothek, um eine Verbindung zu Ihrer DynamoDB-Tabelle herzustellen und die Elemente hinzuzufügen.

   ```
   pip install boto3
   ```

5. Führen Sie das Skript aus, um die Tabelle zu füllen, indem Sie den folgenden Befehl ausführen.

```
python load_sample_data.py
```

   Wenn das Skript erfolgreich läuft, sollte es jedes Element beim Laden auf der Konsole ausgeben und `Data loading completed` melden.

6. Deaktivieren Sie die virtuelle Umgebung, indem Sie den folgenden Befehl ausführen.

```
deactivate
```

7. Sie können überprüfen, ob die Daten in Ihre DynamoDB-Tabelle geladen wurden, indem Sie Folgendes tun:

   a. Öffnen Sie die Seite Elemente untersuchen der DynamoDB-Konsole und wählen Sie Ihre Tabelle (`MyOrderTable`).

   b. Im Bereich Zurückgegebene Elemente sollten Sie die 25 Elemente aus der CSV-Datei sehen, die das Skript der Tabelle hinzugefügt hat.

Die App für geplante Wartungsarbeiten erstellen

Sie können die Ressourcen für diese Beispiel-App Schritt für Schritt mithilfe von AWS-Managementkonsole oder mithilfe von erstellen und bereitstellen. AWS SAM In einer Produktionsumgebung empfehlen wir, ein Infrustracture-as-Code (IaC-) Tool zu verwenden, AWS SAM um wiederholt serverlose Anwendungen bereitzustellen, ohne manuelle Prozesse zu verwenden.

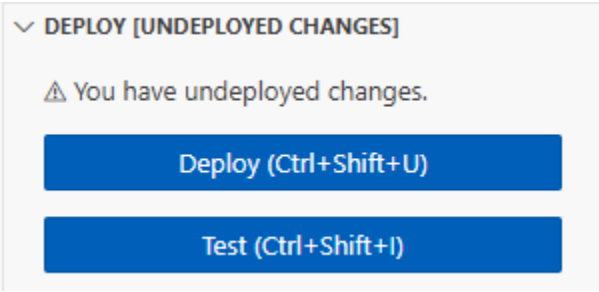
Folgen Sie in diesem Beispiel den Anweisungen auf der Konsole, um zu erfahren, wie Sie jede AWS Ressource separat konfigurieren, oder folgen Sie den AWS SAM Anweisungen, um die App mithilfe von Befehlen schnell bereitzustellen. AWS CLI

Console

Um die Funktion mit dem zu erstellen AWS-Managementkonsole

Erstellen Sie zunächst eine Funktion, die den grundlegenden Startercode enthält. Anschließend ersetzen Sie diesen Code durch Ihren eigenen Funktionscode, indem Sie den Code entweder kopieren und direkt im Lambda-Code-Editor einfügen oder Ihren Code als `.zip`-Paket hochladen. Für diese Aufgabe empfehlen wir, den Code zu kopieren und einzufügen.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Wählen Sie Author from scratch aus.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Function name (Funktionsname) **`ScheduledDBMaintenance`** ein.
   b. Für Laufzeit wählen Sie die neueste Python-Version.
   c. Wählen Sie für Architektur x86_64 aus.
5. Wählen Sie „Event-Funktion erstellen“.
6. Nachdem Ihre Funktion erstellt wurde, können Sie Ihre Funktion mit dem bereitgestellten Funktionscode konfigurieren.
   a. Ersetzen Sie im Bereich Codequelle den Hello World-Code, den Lambda erstellt hat, durch den Python-Funktionscode aus der `lambda_function.py`-Datei, die Sie zuvor gespeichert haben.
   b. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Um den Funktionsspeicher und das Timeout (Konsole) zu konfigurieren

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie im Bereich Allgemeine Konfiguration die Option Bearbeiten aus.
3. Stellen Sie Speicher auf 256 MB und Timeout auf 15 Sekunden ein. Wenn Sie eine große Tabelle mit vielen Datensätzen verarbeiten, z. B. in einer Produktionsumgebung, sollten Sie in Erwägung ziehen, Timeout auf einen höheren Wert festzulegen. Dadurch hat Ihre Funktion mehr Zeit zum Scannen und Bereinigen der Datenbank.
4. Wählen Sie Speichern.

So konfigurieren Sie das Protokollformat (Konsole)

Sie können Lambda-Funktionen so konfigurieren, dass sie Protokolle entweder im unstrukturierten Text- oder im JSON-Format ausgeben. Wir empfehlen, das JSON-Format für Protokolle zu verwenden, um die Protokolldaten einfacher durchsuchen und filtern zu können. Weitere Informationen über die Optionen für die Konfiguration des Lambda-Protokolls finden Sie unter the section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie Überwachungs- und Betriebstools aus.
3. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
4. Wählen Sie für die Konfiguration der Protokollierung JSON aus.
5. Wählen Sie Speichern.

So richten Sie IAM-Berechtigungen ein

Um Ihrer Funktion die erforderlichen Berechtigungen zum Lesen und Löschen von DynamoDB-Elementen zu erteilen, müssen Sie der Ausführungsrolle Ihrer Funktion eine Richtlinie hinzufügen, die die erforderlichen Berechtigungen definiert.

1. Öffnen Sie die Registerkarte Konfiguration und wählen Sie dann in der linken Navigationsleiste Berechtigungen aus.
2. Wählen Sie den Rollennamen unter Ausführungsrolle.
3. Wählen Sie in der IAM-Konsole Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.
4. Verwenden Sie den JSON-Editor und geben Sie die folgenden Richtlinien ein:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Scan",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
```

```
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/MyOrderTable"
        }
    ]
}
```

5. Geben Sie der Richtlinie **DynamoDBCleanupPolicy** einen Namen und erstellen Sie sie dann.

Um den EventBridge Scheduler als Trigger einzurichten (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Ihre Funktion (`ScheduledDBMaintenance`).
3. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).
4. Wählen Sie aus dem Drop-down-Menü für die Triggerquelle die Option Scheduler aus.
5. Konfigurieren Sie den Zeitplan, indem Sie wie folgt vorgehen:
   a. Geben Sie als Namen des Zeitplans ein. **DynamoDBCleanupSchedule**
   b. Behalten Sie für die Zeitplangruppe die Standardeinstellung (`default`) bei.
   c. Wählen Sie als Zeitplantyp die Option Wiederkehrender Zeitplan und geben Sie den folgenden Cron-Ausdruck ein:**cron(0 3 1 * ? *)**.

      Bei der Auswertung wird dieser Cron-Ausdruck am ersten Tag jedes Monats um 03:00 Uhr ausgeführt.
   d. Wählen Sie für Flexibles Zeitfenster die Option Aus.
   e. Lassen Sie das Feld Payload leer.
   f. Wählen Sie für die Ausführungsrolle die Option Neue Rolle für diesen Zeitplan erstellen aus und geben Sie einen Rollennamen ein. EventBridge Der Scheduler erstellt eine neue Richtlinie mit den erforderlichen Berechtigungen, die der Zeitplan benötigt, um Ihre Funktion aufzurufen.
6. Wählen Sie Hinzufügen aus.

AWS SAM

Um die App bereitzustellen mit AWS SAM

1. Navigieren Sie zu dem Ordner, in dem Sie die `template.yaml`-Datei für die App gespeichert haben. Beachten Sie, dass in diesem Beispiel zwei `template.yaml`-Dateien verwendet werden. Vergewissern Sie sich, dass sie in separaten Unterordnern gespeichert sind und dass Sie sich im richtigen Ordner befinden, der die Vorlage zur Erstellung Ihrer App enthält.
2. Kopieren Sie die `lambda_function.py`- und `requirements.txt`-Dateien, die Sie zuvor heruntergeladen haben, in denselben Ordner. Der in der AWS SAM Vorlage angegebene Codespeicherort ist `./`, also der aktuelle Standort. AWS SAM sucht in diesem Ordner nach dem Lambda-Funktionscode, wenn Sie versuchen, die App bereitzustellen.
3. Führen Sie den folgenden Befehl aus.

   ```
   sam build --use-container
   ```

   Dieser Befehl sammelt die Build-Artefakte für die Ressourcen, die Sie bereitstellen möchten und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen. Wenn Sie die `--use-container` Option angeben, wird Ihre Funktion in einem Lambda-like Docker-Container erstellt. Wir verwenden es hier, sodass Sie Python 3.12 nicht auf Ihrem lokalen Computer installiert haben müssen, damit der Build funktioniert.
4. Führen Sie den folgenden Befehl aus, um die in der `template.yaml` Datei angegebenen Lambda- und EventBridge Scheduler-Ressourcen zu erstellen.

   ```
   sam deploy --guided
   ```

   Wenn Sie das `--guided` Kennzeichen verwenden, werden Ihnen Eingabeaufforderungen AWS SAM angezeigt, die Sie durch den Bereitstellungsprozess führen. Geben Sie für diese Bereitstellung ein `Stack name` von **`cron-maintenance-app`** ein und akzeptieren Sie die Standardeinstellungen für alle anderen Optionen mit der Eingabetaste.

   Wenn AWS SAM die Erstellung der Lambda- und EventBridge Scheduler-Ressourcen abgeschlossen ist, sollte die folgende Meldung angezeigt werden.

   ```
   Successfully created/updated stack - cron-maintenance-app in us-west-2
   ```

5. Sie können außerdem überprüfen, ob die Lambda-Funktion erstellt wurde, indem Sie die Seite Funktionen der Lambda-Konsole öffnen. Sie sollten eine Funktion mit dem Namen `ScheduledDBMaintenance` sehen.

Testen der App

Um zu testen, ob Ihr Zeitplan Ihre Funktion korrekt auslöst und ob Ihre Funktion die Datensätze in der Datenbank korrekt bereinigt, können Sie Ihren Zeitplan vorübergehend so ändern, dass er einmal zu einer bestimmten Zeit ausgeführt wird. Sie können dann `sam deploy` erneut ausführen, um Ihren Wiederholungsplan so zurückzusetzen, dass er einmal pro Monat ausgeführt wird.

Um die Anwendung mit dem auszuführen AWS-Managementkonsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Ihre Funktion (`ScheduledDBMaintenance`).
3. Wählen Sie im Bereich Trigger den `DynamoDBCleanupSchedule` Auslöser aus.
4. Wählen Sie Bearbeiten aus.
5. Wählen Sie als Zeitplantyp die Option One-time Zeitplan aus.
6. Stellen Sie Ihre Aufrufzeit auf ein paar Minuten ein und wählen Sie dann Speichern.

Nachdem der Zeitplan ausgeführt und sein Ziel aufgerufen wurde, führen Sie das `test_app.py`-Skript aus, um zu überprüfen, ob Ihre Funktion erfolgreich alle alten Datensätze aus der DynamoDB-Tabelle entfernt hat.

Um zu überprüfen, ob alte Datensätze mit einem Python-Skript gelöscht wurden

1. Navigieren Sie in Ihrer Befehlszeile zu dem Ordner, in dem Sie `test_app.py` gespeichert haben.
2. Führen Sie das Skript aus.

   ```
   python test_app.py
   ```

   Bei Erfolg wird die folgende Ausgabe angezeigt.

   ```
   Total number of old records: 0
   ```

Nächste Schritte

Sie können den EventBridge Scheduler-Zeitplan jetzt an Ihre speziellen Anwendungsanforderungen anpassen. EventBridge Scheduler unterstützt die folgenden Zeitplanausdrücke: Cron, Rate und One-Time Schedules.

Weitere Informationen zu EventBridge Scheduler-Zeitplanausdrücken finden Sie unter Zeitplantypen im EventBridge Scheduler-Benutzerhandbuch. Zugriffsverwaltung im IAM-Benutzerhandbuch

Programmiersprachen

Lambda Event Functions (früher bekannt als Lambda-Funktionen) unterstützen eine Vielzahl gängiger Programmiersprachen. Jede hat ein idiomatisches Programmiererlebnis, das auf diese Sprache zugeschnitten ist.

Jede Programmiersprache wird von einer entsprechenden Runtime unterstützt, die die sprachspezifische Ausführungsumgebung für Ihre Funktion bereitstellt.

- the section called “Bauen mit Node.js”
- the section called “Bauen mit TypeScript”
- the section called “Erstellen mit Python”
- the section called “Erstellen mit Ruby”
- the section called “Erstellen mit Java”
- the section called “Erstellen mit Go”
- the section called “Erstellen mit C#”
- the section called “Bauen mit PowerShell”
- the section called “Erstellen mit Rust”

Erstellen von Lambda-Eventfunktionen mit Node.js

Sie können rennen JavaScript codieren mit Node.js in AWS Lambda Event-Funktionen. Lambda bietet Laufzeiten zum Node.js die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Umgebung ausgeführt, die Folgendes umfasst AWS SDK für JavaScript, mit Anmeldeinformationen von einem AWS Identity and Access Management (IAM) -Rolle, die Sie verwalten. Um mehr über die SDK-Versionen zu erfahren, die im Lieferumfang des enthalten sind Node.js Laufzeiten finden Sie unter the section called “Runtime-included SDK-Versionen”.

Lambda unterstützt Folgendes Node.js Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 26	`nodejs26.x`	Amazon Linux 2023	Nicht geplant	Nicht geplant	Nicht geplant
Node.js 24	`nodejs24.x`	Amazon Linux 2023	30. April 2028	1. Juni 2028	1. Juli 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	30. April 2027	29. Juli 2027	31. August 2027

Um eine zu erstellen Node.js Event-Funktion

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Event-Funktion erstellen.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Laufzeit: Wählen Node.js 24.x.
4. Wählen Event-Funktion erstellen.

Die Konsole erstellt eine Lambda-Ereignisfunktion mit einer einzigen Quelldatei namens `index.mjs`. Mit dem integrierten Code-Editor können Sie diese Datei bearbeiten und weitere Dateien hinzufügen. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren. Um Ihren Code auszuführen, wählen Sie anschließend im Abschnitt TESTEREIGNISSE die Option Testereignis erstellen aus.

Die `index.mjs`-Datei exportiert eine Funktion mit dem Namen `handler`, die ein Ereignisobjekt und ein Kontext-Objekt übernimmt. Dies ist die Handler-Funktion, die bei einem Aufruf der Funktion von Lambda aufgerufen wird. Das Node.js Die Funktionslaufzeit ruft Aufrufereignisse von Lambda ab und übergibt sie an den Handler. In der Konfiguration der Funktion lautet der Wert für den Handler `index.handler`.

Wenn Sie Ihren Funktionscode speichern, erstellt die Lambda-Konsole ein Bereitstellungspaket für das ZIP-Dateiarchiv. Wenn Sie Ihren Funktionscode außerhalb der Konsole (mit einer IDE) entwickeln, müssen Sie ein Bereitstellungspaket erstellen, um Ihren Code in die Lambda-Funktion hochzuladen.

Die Funktionslaufzeit übergibt neben dem Aufrufereignis ein Context-Objekt an den Handler. Das Context-Objekt enthält zusätzliche Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung. Weitere Informationen erhalten Sie über die Umgebungsvariablen.

Ihre Lambda-Funktion enthält eine CloudWatch Logs, Protokollgruppe. Die Funktion Runtime sendet Details zu jedem Aufruf an CloudWatch Loggt. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- Runtime-included SDK-Versionen
- Verwenden von Keepalive für TCP-Verbindungen
- CA-Zertifikat wird geladen
- Experimentell Node.js Eigenschaften
- Definieren Sie den Lambda-Ereignisfunktionshandler in Node.js
- Stellen Sie Node.js Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit
- Stellen Sie Node.js Lambda-Ereignisfunktionen mit Container-Images bereit
- Arbeiten mit Ebenen für Node.js Lambda-Ereignisfunktionen
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Node.js Ereignisfunktion
- Node.js Lambda-Ereignisfunktionen protokollieren und überwachen
- Node.js Code instrumentieren in AWS Lambda

Runtime-included SDK-Versionen

Alle unterstützte Lambda Node.js Laufzeiten enthalten eine bestimmte Nebenversion von AWS SDK für JavaScript v3, nicht der neueste Version. Welche Nebenversion in der Laufzeit enthalten ist, hängt von der Laufzeitversion und Ihrer AWS-Region ab. Um die spezifische Version des SDK zu finden, die in der von Ihnen verwendeten Laufzeit enthalten ist, erstellen Sie eine Lambda-Ereignisfunktion mit dem folgenden Code.

Example index.mjs

```
import packageJson from '@aws-sdk/client-s3/package.json' with { type: 'json' };

export const handler = async () => ({ version: packageJson.version });
```

Dies gibt eine Antwort im folgenden Format zurück:

```
{
  "version": "3.632.0"
}
```

Weitere Informationen finden Sie unter Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler.

Verwenden von Keepalive für TCP-Verbindungen

Die Standardeinstellung Node.js HTTP/HTTPS Der Agent erstellt für jede neue Anfrage eine neue TCP-Verbindung. Um die Kosten für den Aufbau neuer Verbindungen zu vermeiden, ist Keep-Alive standardmäßig in allen aktiviert unterstützt Node.js Laufzeiten. Keep-alive kann die Anforderungszeiten für Lambda-Funktionen reduzieren, die mithilfe des SDK mehrere API-Aufrufe tätigen.

Informationen zur Deaktivierung von Keep-Alive finden Sie unter Wiederverwendung von Verbindungen mit Keep-Alive in Node.js in der AWS SDK für JavaScript 3.x Entwicklerhandbuch. Weitere Informationen zur Verwendung von Keep-Alive finden Sie unter HTTP-Keep-Alive ist in Modular standardmäßig aktiviert AWS SDK für JavaScript auf der AWS Blog mit Tools für Entwickler.

CA-Zertifikat wird geladen

Für Node.js Runtime-Versionen bis zu Node.js 18, Lambda wird automatisch geladen Amazon-specific CA-Zertifikate (Certificate Authority), um es Ihnen zu erleichtern, Funktionen zu erstellen, die mit anderen interagieren AWS-Services. Lambda enthält beispielsweise die Amazon-RDS-Zertifikate, die für die Validierung des in Ihrer Amazon-RDS-Datenbank installierten Server-Identitätszertifikats erforderlich sind. Dieses Verhalten kann sich bei Kaltstarts auf die Leistung auswirken.

Beginnend mit Node.js 20, Lambda lädt standardmäßig keine zusätzlichen CA-Zertifikate mehr. Das Node.js 20 Runtime enthält eine Zertifikatsdatei mit allen Amazon CA-Zertifikaten, die sich unter `/var/runtime/ca-cert.pem`. Um dasselbe Verhalten wiederherzustellen von Node.js 18 und frühere Laufzeiten stellen Sie ein `NODE_EXTRA_CA_CERTS` Umgebungsvariable zu `/var/runtime/ca-cert.pem`.

Für eine optimale Leistung empfehlen wir, nur die Zertifikate, die Sie benötigen, mit Ihrem Bereitstellungspaket zu bündeln und sie über das `NODE_EXTRA_CA_CERTS` Umgebungsvariable. Die Zertifikatsdatei sollte aus einem oder mehreren vertrauenswürdigen Stammzertifikaten oder Zertifikaten von Zwischenzertifizierungsstellen im PEM-Format bestehen. Fügen Sie für RDS die erforderlichen Zertifikate beispielsweise zusammen mit Ihrem Code als `certificates/rds.pem` hinzu. Laden Sie dann die Zertifikate, indem Sie `NODE_EXTRA_CA_CERTS` auf `/var/task/certificates/rds.pem` festlegen.

Experimentell Node.js Eigenschaften

Der Upstream Node.js Sprachversionen ermöglichen standardmäßig einige experimentelle Funktionen. Lambda deaktiviert diese Features, um die Laufzeitstabilität und eine konsistente Leistung zu gewährleisten. In der folgenden Tabelle sind die experimentellen Features aufgeführt, die Lambda deaktiviert.

Experimentelle Features	Unterstützt Node.js Versionen	Node.js von Lambda angewendete Flagge	Lambda-Flag zum erneuten Aktivieren
Support für den Import von Modulen mithilfe von „require“ in ES-Modulen	Node.js 20, Node.js 22, Node.js 24	`--no-experimental-require-module`	`--experimental-require-module`
Support für die automatische Erkennung von ES- und CommonJS-Modulen	Node.js 22, Node.js 24	`--no-experimental-detect-module`	`--experimental-detect-module`

Um ein deaktiviertes experimentelles Feature zu aktivieren, setzen Sie das Reaktivierungs-Flag in der Umgebungsvariablen „`NODE_OPTIONS`“. Beispiel: Um die Unterstützung für „require“ in ES-Modulen zu aktivieren, setzen Sie „`NODE_OPTIONS`“ auf „`--experimental-require-module`“. Lambda erkennt diese Überschreibung und entfernt das entsprechende Deaktivierungs-Flag.

Important

Die Verwendung experimenteller Features kann zu Instabilität und Leistungsproblemen führen. Diese Funktionen könnten in Zukunft geändert oder entfernt werden Node.js Versionen. Funktionen, die experimentelle Features nutzen, sind nicht für das Lambda Service Level Agreement (SLA) oder AWS Support qualifiziert.

Definieren Sie den Lambda-Ereignisfunktionshandler in Node.js

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Ereignisfunktionshandlern arbeiten Node.js, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Node.js Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdateiquittung erstellt und diese Datei in einem Amazon Simple Storage Service (Amazon S3) -Bucket ablegt. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Themen

- Richten Sie Ihr Node.js Handler-Projekt ein
- Beispiel für einen Node.js Lambda-Ereignisfunktionscode
- CommonJS- und ES-Module
- Node.js Initialisierung
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Gültige Handlermuster für Funktionen Node.js
- Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für Node.js Lambda-Ereignisfunktionen

Richten Sie Ihr Node.js Handler-Projekt ein

Es gibt mehrere Möglichkeiten, ein Node.js Lambda-Projekt zu initialisieren. Sie können beispielsweise ein Node.js Standardprojekt erstellen`npm`, indem Sie eine AWS SAM Anwendung erstellen oder eine Anwendung erstellen.AWS CDK

So erstellen Sie das Projekt mit `npm`:

```
npm init
```

Dieser Befehl initialisiert Ihr Projekt und generiert eine `package.json`-Datei, die die Metadaten und Abhängigkeiten Ihres Projekts verwaltet.

Ihr Funktionscode befindet sich in einer `.js .mjs` JavaScript OR-Datei. Im folgenden Beispiel nennen wir diese Datei `index.mjs`, weil sie einen ES-Modul-Handler verwendet. Lambda unterstützt sowohl ES-Modul- als auch CommonJS-Handler. Weitere Informationen finden Sie unter CommonJS- und ES-Module.

Ein typisches Node.js Lambda-Funktionsprojekt folgt dieser allgemeinen Struktur:

```
/project-root
  ### index.mjs – Contains main handler
  ### package.json – Project metadata and dependencies
  ### package-lock.json – Dependency lock file
  ### node_modules/ – Installed dependencies
```

Beispiel für einen Node.js Lambda-Ereignisfunktionscode

Das folgende Beispiel für einen Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example index.mjs Lambda-Ereignisfunktion

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
export const handler = async(event) => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
```

```
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error.message}`);
        throw error;
    }
};

/**
 * Helper function to upload receipt to S3
 * @param {string} bucketName - The S3 bucket name
 * @param {string} key - The S3 object key
 * @param {string} receiptContent - The content to upload
 * @returns {Promise<void>}
 */
async function uploadReceiptToS3(bucketName, key, receiptContent) {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error.message}`);
    }
}
```

Diese `index.mjs`-Datei enthält die folgenden Abschnitte des Codes:

- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt, z. B. AWS SDK-Clients.
- `const s3Client`-Deklaration: Hiermit wird ein Amazon-S3-Client außerhalb der Handler-Funktion initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen und der Client wird für die Wiederverwendung über mehrere Aufrufe hinweg aufbewahrt.
- JSDoc-Kommentarblock: Definieren Sie die Eingabe- und Ausgabetypen für Ihren Handler mithilfe von JSDoc-Annotationen.
- `export const handler`: Dies ist die Haupt-Handler-Funktion, die Lambda aufruft. Geben Sie beim Bereitstellen Ihrer Funktion `index.handler` für die Handler-Eigenschaft an. Der Wert der Eigenschaft `Handler` besteht aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt.
- `uploadReceiptToS3`-Funktion: Dies ist eine Hilfsfunktion, auf die von der Haupt-Handler-Funktion verwiesen wird.

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

CommonJS- und ES-Module

Node.js unterstützt zwei Modulsysteme: CommonJS- und ECMAScript-Module (ES-Module). Lambda empfiehlt die Verwendung von ES-Modulen, da sie Await auf oberster Ebene unterstützen, wodurch asynchrone Aufgaben während der Initialisierung der Ausführungsumgebung abgeschlossen werden können.

Node.js behandelt Dateien mit einer `.cjs` Dateinamenerweiterung als CommonJS-Module, während eine `.mjs` Erweiterung ES-Module bezeichnet. Node.js Behandelt Dateien mit der `.js` Dateinamenerweiterung standardmäßig als CommonJS-Module. Sie können so konfigurieren Node.js , dass `.js` Dateien als ES-Module behandelt werden, indem Sie das `type` as `module` in der `package.json` Funktionsdatei angeben. Sie können Node.js in Lambda so konfigurieren, dass automatisch erkannt wird, ob eine `.js` Datei als CommonJS- oder als ES-Modul behandelt werden soll, indem Sie der `NODE_OPTIONS` Umgebungsvariablen das `–experimental-detect-module` Flag hinzufügen. Weitere Informationen finden Sie unter Experimentelle Node.js Funktionen.

Die folgenden Beispiele zeigen Funktionshandler, die sowohl mit ES-Modulen als auch mit CommonJS-Modulen geschrieben wurden. Die übrigen Beispiele auf dieser Seite verwenden alle ES-Module.

ES module example

Example– ES-Modul-Handler

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module example

Example– CommonJS-Modul-Handler

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

Node.js Initialisierung

Node.js verwendet ein nicht blockierendes I/O Modell, das effiziente asynchrone Operationen mithilfe einer Ereignisschleife unterstützt. Wenn Node.js die Funktion beispielsweise einen Netzwerkaufruf tätigt, verarbeitet sie weiterhin andere Operationen, ohne eine Netzwerkantwort zu blockieren. Wenn die Netzwerkantwort empfangen wird, wird sie in die Rückruf-Warteschlange gestellt. Aufgaben aus der Warteschlange werden verarbeitet, wenn die aktuelle Aufgabe abgeschlossen ist.

Lambda empfiehlt die Verwendung von Await auf oberster Ebene, damit asynchrone Aufgaben, die während der Initialisierung der Ausführungsumgebung initiiert wurden, während der Initialisierung abgeschlossen werden. Asynchrone Aufgaben, die während der Initialisierung nicht abgeschlossen werden, werden normalerweise beim ersten Funktionsaufruf ausgeführt. Dies kann zu unerwartetem Verhalten oder Fehlern führen.

Beispielsweise könnte Ihre Funktionsinitialisierung einen Netzwerkaufruf ausführen, um einen Parameter aus dem AWS Parameterspeicher abzurufen. Wenn diese Aufgabe während der Initialisierung nicht abgeschlossen wird, kann der Wert während eines Aufrufs Null sein. Es kann auch zu Verzögerungen zwischen Initialisierung und Aufruf kommen, was bei zeitkritischen Vorgängen zu Fehlern führen kann. Insbesondere AWS Serviceabrufe können sich auf zeitkritische Anforderungssignaturen stützen, was dazu führt, dass Serviceabrufe fehlschlagen, wenn der Anruf während der Initialisierungsphase nicht abgeschlossen wird.

Das Abschließen von Aufgaben während der Initialisierung verbessert in der Regel die Kaltstartleistung und die Leistung beim ersten Aufrufen, wenn Provisioned Concurrency verwendet wird. Weitere Informationen finden Sie in unserem Blogbeitrag Using Node.js ES modules and top level await in. AWS Lambda

Namenskonventionen für Handler

Wenn Sie eine Funktion konfigurieren, besteht der Wert der Handler-Einstellung aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt. Der Standardwert für Funktionen, die in der Konsole erstellt werden und für die Beispiele in diesem Handbuch ist `index.handler`. Dies deutet auf die `handler`-Methode hin, die aus der `index.js`- oder `index.mjs`-Datei wurde.

Wenn Sie eine Funktion in der Konsole mit einem anderen Dateinamen oder Funktionshandlernamen erstellen, müssen Sie den Standardhandlernamen bearbeiten.

So ändern Sie den Funktionshandlernamen (Konsole)

1. Öffnen Sie die Seite Functions (Funktionen) der Lambda-Konsole und wählen Sie eine Funktion aus.
2. Wählen Sie die Registerkarte Code (Code).
3. Scrollen Sie nach unten zum Bereich Laufzeiteinstellungen und wählen Sie Bearbeiten.
4. Geben Sie unter Handler den neuen Namen für Ihren Funktionshandler ein.
5. Wählen Sie Speichern.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in arbeiten Node.js, können Sie die erwartete Form des Eingabeereignisses mithilfe von JSDoc-Annotationen definieren. In diesem Beispiel definieren wir die Eingabestruktur im JSDoc-Kommentar des Handlers:

```
/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
```

Nachdem Sie diese Typen in Ihrem JSDoc-Kommentar definiert haben, können Sie direkt in Ihrem Code auf die Felder des Ereignisobjekts zugreifen. Zum Beispiel holt `event.order_id` den Wert von `order_id` aus der ursprünglichen Eingabe.

Gültige Handlermuster für Funktionen Node.js

Wir empfehlen, den Funktionshandler async/awaitzu deklarieren, anstatt Callbacks zu verwenden. Async/await ist eine präzise und lesbare Methode, asynchronen Code zu schreiben, ohne dass verschachtelte Callbacks oder verkettete Zusagen erforderlich sind. Mit können Sie Code schreiben async/await, der sich wie synchroner Code liest, aber trotzdem asynchron ist und nicht blockiert.

asynchrone Funktionshandler (empfohlen)

Das `async`-Schlüsselwort kennzeichnet eine Funktion als asynchron, und das `await`-Schlüsselwort unterbricht die Ausführung der Funktion, bis `Promise` aufgelöst ist. Der Handler akzeptiert die folgenden Argumente:

- `event`: Enthält die an Ihre Funktion übergebenen Eingabedaten.
- `context`: Enthält Informationen über den Aufruf, die Funktion und die Ausführungsumgebung. Weitere Informationen finden Sie unter Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Node.js Ereignisfunktion.

Hier sind die gültigen Signaturen für das async/await Muster:

```
export const handler = async (event) => { };
```

```
export const handler = async (event, context) => { };
```

Handler für synchrone Funktionen

Wenn Ihre Funktion keine asynchronen Aufgaben ausführt, können Sie einen synchronen Funktionshandler verwenden, der eine der folgenden Funktionssignaturen verwendet:

```
export const handler = (event) => { };
```

```
export const handler = (event, context) => { };
```

Handler für Antwort-Streaming-Funktionen

Lambda unterstützt Response-Streaming mit Node.js. Handler für Antwort-Streaming-Funktionen verwenden den `awslambda.streamifyResponse()` Decorator und verwenden 3 Parameter:`event`, `responseStream` und. `context` Die Funktionssignatur lautet:

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 context) => { });
```

Weitere Informationen finden Sie unter Antwortstreaming für Lambda-Ereignisfunktionen.

Callback-based Funktionshandler

> **Note**
>
> Callback-based Funktionshandler werden nur bis zu Node.js 22 unterstützt. Ab Node.js 24 sollten asynchrone Aufgaben mit asynchronen Funktionshandlern implementiert werden.

Callback-based Funktionshandler müssen die Argumente Event, Kontext und Callback verwenden. Beispiel:

```
export const handler = (event, context, callback) => { };
```

Die Callback-Funktion erwartet eine Antwort `Error` und diese muss es sein. JSON-serializable Die Funktion wird so lange ausgeführt, bis die Ereignisschleife leer ist oder ein Timeout auftritt. Die Antwort wird erst an den Aufrufer gesendet, wenn alle Ereignisschleifenaufgaben abgeschlossen sind. Wenn eine Zeitüberschreitung der Funktion auftritt, wird stattdessen ein Fehler zurückgegeben. Sie können die Laufzeit so konfigurieren, dass die Antwort sofort gesendet wird, indem Sie WaitsForEmptyEventLoopcontext.callback auf false setzen.

Example– HTTP-Anforderung mit callback

Die folgende Beispielfunktion prüft eine URL und gibt den Statuscode an den Aufrufer zurück.

```
import https from "https";
let url = "https://aws.amazon.com/";

export const handler = (event, context, callback) => {
  https.get(url, (res) => {
    callback(null, res.statusCode);
  }).on("error", (e) => {
    callback(Error(e));
  });
};
```

Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für JavaScript. Alle unterstützten Node.js Lambda-Laufzeiten enthalten das SDK für JavaScript Version 3.

Wir empfehlen jedoch dringend, dass Sie die benötigten AWS SDK-Clients in Ihr Bereitstellungspaket aufnehmen. Dadurch wird die Abwärtskompatibilität bei zukünftigen Lambda-Laufzeit-Updates maximiert. Verlassen Sie sich nur dann auf das von der Laufzeit bereitgestellte SDK, wenn Sie keine zusätzlichen Pakete hinzufügen können (z. B. wenn Sie den Code-Editor der Lambda-Konsole oder Inline-Code in einer AWS CloudFormation Vorlage verwenden).

Um Ihrer Funktion SDK-Abhängigkeiten hinzuzufügen, verwenden Sie den `npm install`-Befehl für die spezifischen SDK-Clients, die Sie benötigen. Im Beispielcode haben wir den Amazon-S3-Client verwendet. Fügen Sie diese Abhängigkeit hinzu, indem Sie den folgenden Befehl in dem Verzeichnis ausführen, das Ihre `package.json`-Datei enthält:

```
npm install @aws-sdk/client-s3
```

Importieren Sie im Funktionscode den Client und die Befehle, die Sie benötigen, wie in der Beispielfunktion gezeigt:

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Initialisieren Sie dann einen Amazon-S3-Client:

```
const s3Client = new S3Client();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn verwenden, um API-Aufrufe für diesen Dienst zu tätigen. AWS Der Beispielcode ruft die Amazon S3 PutObjectS3-API-Aktion wie folgt auf:

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
```

```
    Body: receiptContent
});
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe von `process.env` auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit den folgenden Codezeilen auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb des Handlers. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis ausführt, und der Client bleibt für die Wiederverwendung bei allen Aufrufen verfügbar.

Bewährte Codemethoden für Node.js Lambda-Ereignisfunktionen

Befolgen Sie diese Richtlinien, wenn Sie Lambda-Funktionen erstellen:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Node.js und Python-Laufzeiten gehören dazu die AWS SDKs. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.

- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates auf die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel

sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Stellen Sie Node.js Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit

Der Code Ihrer AWS Lambda Funktion besteht aus einer .js- oder .mjs-Datei, die den Handlercode Ihrer Funktion sowie alle zusätzlichen Pakete und Module enthält, von denen Ihr Code abhängt. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Dieses Paket kann entweder ein ZIP-Dateiarchiv oder ein Container-Image sein. Weitere Informationen zur Verwendung von Container-Images mit Node.js finden Sie unter Bereitstellen von Node.js Lambda-Ereignisfunktionen mit Container-Images.

Zum Erstellen des Bereitstellungspakets für ein ZIP-Dateiarchiv können Sie ein integriertes Dienstprogramm für ZIP-Dateien Ihres Befehlszeilen-Tools oder ein anderes Dienstprogramm für ZIP-Dateien verwenden, wie 7zip. In den Beispiele in den folgenden Abschnitten wird davon ausgegangen, dass Sie ein `zip`-Befehlszeilen-Tool in einer Linux- oder MacOS-Umgebung verwenden. Um dieselben Befehle in Windows zu verwenden, können Sie das Windows-Subsystem für Linux installieren, um eine Windows-integrated Version von Ubuntu und Bash zu erhalten.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

Themen

- Laufzeitabhängigkeiten in Node.js
- ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen
- ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen
- Erstellen Sie eine Node.js Ebene für Ihre Abhängigkeiten
- Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken
- Node.js Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Laufzeitabhängigkeiten in Node.js

Für Lambda-Ereignisfunktionen, die die Node.js Laufzeit verwenden, kann eine Abhängigkeit ein beliebiges Node.js Modul sein. Die Node.js Laufzeit umfasst eine Reihe gängiger Bibliotheken sowie eine Version von. AWS SDK für JavaScript Alle unterstützten Node.js Laufzeiten enthalten Version 3 des SDK. Um Version 2 des SDK zu verwenden, müssen Sie das SDK zu Ihrem ZIP-Dateibereitstellungspaket hinzufügen. Die spezifische Version des SDK, die in der von Ihnen verwendeten Laufzeit enthalten ist, finden Sie unter the section called “Runtime-included SDK-Versionen”.

Lambda aktualisiert die SDK-Bibliotheken während der Node.js Laufzeit regelmäßig, sodass sie die neuesten Funktionen und Sicherheitsupgrades enthalten. Lambda wendet auch Sicherheitspatches und Updates auf die anderen in der Laufzeit enthaltenen Bibliotheken an. Um die volle Kontrolle über die Abhängigkeiten in Ihrem Paket zu haben, können Sie Ihrem Bereitstellungspaket Ihre bevorzugte Version einer beliebigen in die Laufzeit eingeschlossenen Abhängigkeit hinzufügen. Wenn Sie beispielsweise eine bestimmte Version des SDK für verwenden möchten JavaScript, können Sie sie als Abhängigkeit in Ihre ZIP-Datei aufnehmen. Weitere Informationen zum Hinzufügen von in die Laufzeit eingeschlossene Abhängigkeiten zu Ihrer ZIP-Datei finden Sie unter Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken.

Im Rahmen des AWS -Modells der geteilten Verantwortung sind Sie für die Verwaltung aller Abhängigkeiten in den Bereitstellungspaketen Ihrer Funktionen verantwortlich. Dies beinhaltet das Durchführen von Updates und Sicherheitspatches. Zum Aktualisieren von Abhängigkeiten im Bereitstellungspaket Ihrer Funktion erstellen Sie zunächst eine neue ZIP-Datei und laden Sie diese dann in Lambda hoch. Weitere Informationen finden Sie unter ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen und Node.js Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen

Wenn Ihr Funktionscode außer den in der Lambda-Laufzeit enthaltenen Bibliotheken keine Abhängigkeiten hat, enthält Ihre ZIP-Datei nur die `index.mjs` Oder-Datei mit dem `index.js` Handlercode Ihrer Funktion. Erstellen Sie mit Ihrem bevorzugten ZIP-Programm eine ZIP-Datei mit Ihrer `index.js`- oder `index.mjs`-Datei im Stammverzeichnis. Wenn sich die Datei, die Ihren Handler-Code enthält, nicht im Stammverzeichnis Ihrer ZIP-Datei befindet, kann Lambda Ihren Code nicht ausführen.

Informationen zum Bereitstellen Ihrer ZIP-Datei zum Erstellen einer neuen Lambda-Funktion oder Aktualisieren einer vorhandenen Funktion, finden Sie unter Node.js Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen

Wenn Ihr Funktionscode von Paketen oder Modulen abhängt, die nicht in der Node.js Lambda-Laufzeit enthalten sind, können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer .zip-Datei hinzufügen oder eine Lambda-Schicht verwenden. Die Anweisungen in diesem Abschnitt zeigen Ihnen, wie Sie Ihre Abhängigkeiten in Ihr ZIP-Bereitstellungspaket aufnehmen. Anweisungen zum Einschließen Ihrer Abhängigkeiten in eine Ebene finden Sie unter the section called “Erstellen Sie eine Node.js Ebene für Ihre Abhängigkeiten”.

Die folgenden CLI-Befehle erstellen Sie eine ZIP-Datei mit dem Namen `my_deployment_package.zip`, welche die `index.js`- oder `index.mjs`-Datei mit dem Handler-Code Ihrer Funktion und die zugehörigen Abhängigkeiten enthält. Im Beispiel installieren Sie Abhängigkeiten mit dem npm-Paketmanager.

Erstellen des Bereitstellungspakets

1. Navigieren Sie zum Projektverzeichnis, das Ihre Quellcodedatei `index.js` oder `index.mjs` enthält. In diesem Beispiel trägt das Verzeichnis den Namen `my_function`.

   ```
   cd my_function
   ```

2. Installieren Sie die erforderlichen Bibliotheken Ihrer Funktion im Verzeichnis `node_modules` mit dem Befehl `npm install`. In diesem Beispiel installieren Sie das AWS X-Ray-SDK für Node.js.

   ```
   npm install aws-xray-sdk
   ```

   Dabei wird eine Ordnerstruktur erstellt, die etwa wie folgt aussieht:

   ```
   ~/my_function
   ### index.mjs
   ### node_modules
       ### async
       ### async-listener
       ### atomic-batcher
       ### aws-sdk
       ### aws-xray-sdk
       ### aws-xray-sdk-core
   ```

   Sie können Ihrem Bereitstellungspaket auch benutzerdefinierte Module hinzufügen, die Sie selbst erstellen. Erstellen Sie ein Verzeichnis unter `node_modules` mit dem Namen Ihres Moduls und speichern Sie Ihre benutzerdefinierten geschriebenen Pakete dort.

3. Erstellen Sie im Stammverzeichnis eine ZIP-Datei mit den Inhalten Ihres Projektordners. Benutzen Sie die `r` (rekursive) Option, um sicherzustellen, dass der zip-Befehl die Unterordner komprimiert.

   ```
   zip -r my_deployment_package.zip .
   ```

Erstellen Sie eine Node.js Ebene für Ihre Abhängigkeiten

Die Anweisungen in diesem Abschnitt zeigen Ihnen, wie Sie Ihre Abhängigkeiten in eine Ebene einschließen. Anweisungen zum Einschließen Ihrer Abhängigkeiten in Ihr Bereitstellungspaket finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen”. Weitere Informationen zum Erstellen und Verwalten von Node.js Ebenen finden Sie unterthe section called “Layer”.

Wenn Sie einer Funktion eine Ebene hinzufügen, lädt Lambda den Ebeneninhalt in das Verzeichnis `/opt` der Ausführungsumgebung. Für jede Lambda-Laufzeit enthält die Variable `PATH` bereits spezifische Ordnerpfade innerhalb des Verzeichnisses `/opt`. Um sicherzustellen, dass Lambda den Inhalt Ihrer Ebene aufnimmt, müssen die Abhängigkeiten Ihrer ZIP-Datei in einem der folgenden Ordnerpfade enthalten sein:

- `nodejs/node_modules`
- `nodejs/node18/node_modules (NODE_PATH)`
- `nodejs/node20/node_modules (NODE_PATH)`
- `nodejs/node22/node_modules (NODE_PATH)`

Die Struktur Ihrer Ebene-ZIP-Datei könnte beispielsweise wie folgt aussehen:

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Darüber hinaus erkennt Lambda automatisch alle Bibliotheken im `/opt/lib`-Verzeichnis und alle Binärdateien im `/opt/bin`-Verzeichnis. Um sicherzustellen, dass Lambda Ihren Ebeneninhalt korrekt findet, können Sie auch eine Ebene mit der folgenden Struktur erstellen:

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

Nachdem Sie Ihre Ebene gebündelt haben, sehen Sie sich the section called “Erstellen und Löschen von Ebenen” und the section called “Hinzufügen von Ebenen” an, um die Einrichtung Ihrer Ebene abzuschließen.

Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken

Die Node.js Laufzeit umfasst eine Reihe gängiger Bibliotheken sowie eine Version von AWS SDK für JavaScript. Wenn Sie eine andere Version einer in die Laufzeit eingeschlossenen Bibliothek verwenden möchten, können Sie sie mit Ihrer Funktion bündeln oder sie als Abhängigkeit zu Ihrem Bereitstellungspaket hinzufügen. Sie können beispielsweise eine andere Version des SDK verwenden, indem Sie es Ihrem ZIP-Bereitstellungspaket hinzufügen. Sie können sie auch in eine Lambda-Ebene für Ihre Funktion aufnehmen.

Wenn Sie in Ihrem Code eine `import` `require` OR-Anweisung verwenden, durchsucht die Node.js Runtime die Verzeichnisse im `NODE_PATH` Pfad, bis sie das Modul findet. Standardmäßig ist der erste Speicherort, den die Laufzeit durchsucht, das Verzeichnis, in das Ihr ZIP-Bereitstellungspaket entpackt und bereitgestellt wird (`/var/task`). Wenn Sie eine Version einer Bibliothek, die in der Laufzeit enthalten ist, in Ihr Bereitstellungspaket aufnehmen, hat diese Version Vorrang vor der Version, die in der Runtime enthalten ist. Abhängigkeiten in Ihrem Bereitstellungspaket haben ebenfalls Vorrang vor Abhängigkeiten in Ebenen.

Wenn Sie einer Ebene eine Abhängigkeit hinzufügen, extrahiert Lambda diese in `/opt/nodejs/nodexx/node_modules`, wobei `nodexx` die von Ihnen verwendete Version der Laufzeitumgebung ist. Im Suchpfad hat dieses Verzeichnis Vorrang vor dem Verzeichnis mit den zur Laufzeit enthaltenen Bibliotheken (`/var/lang/lib/node_modules`). Bibliotheken in Funktionsebenen haben daher Vorrang vor Versionen, die in der Laufzeit enthalten sind.

Sie können den vollständigen Suchpfad für Ihre Lambda-Funktion sehen, indem Sie die folgende Codezeile hinzufügen:

```
console.log(process.env.NODE_PATH)
```

Fügen Sie Abhängigkeiten alternativ auch in einem separaten Ordner in Ihrem ZIP-Paket hinzu. Fügen Sie beispielsweise eine benutzerdefinierte Version zu einem Ordner in Ihrem ZIP-Paket mit dem Namen `common` hinzu. Wird Ihr ZIP-Paket entpackt und bereitgestellt, wird dieser Ordner im Verzeichnis „`/var/task`“ abgelegt. Um eine Abhängigkeit von einem Ordner in Ihrem ZIP-Bereitstellungspaket in Ihrem Code zu verwenden, verwenden Sie eine `import { } from`- oder `const { } = require()`-Anweisung, je nachdem, ob Sie die CJS- oder die ESM-Modulauflösung verwenden. Beispiel:

```
import { myModule } from './common'
```

Wenn Sie Ihren Code mit `esbuild`, `rollup` oder ähnlichem bündeln, werden die von Ihrer Funktion verwendeten Abhängigkeiten in einer oder mehreren Dateien gebündelt. Wir empfehlen, diese Methode zu verwenden, um Abhängigkeiten wann immer möglich zu vergeben. Im Vergleich zum Hinzufügen von Abhängigkeiten zu Ihrem Bereitstellungspaket führt das Bündeln Ihres Codes aufgrund der Reduzierung der Betriebsabläufe zu einer Leistungssteigerung. I/O

Node.js Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Sie können Ihr .zip-Paket mithilfe der Lambda-Konsole, der und der AWS Command Line Interface Lambda-API bereitstellen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Create Event Function aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.

c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.

4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie „Event-Funktion erstellen“. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:

   a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.

   b. Klicken Sie auf Open.

   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

ZIP-Dateifunktionen mithilfe des Konsolencode-Editors aktualisieren

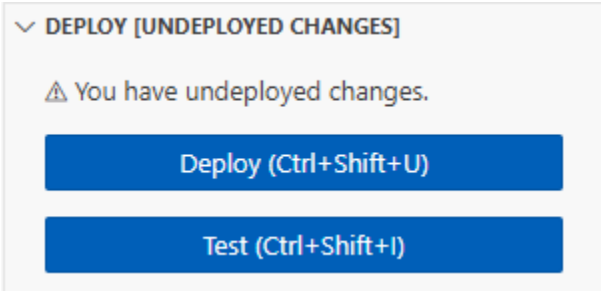
Für einige Funktionen mit ZIP-Bereitstellungspaketen können Sie Ihren Funktionscode direkt mit dem in der Lambda-Konsole integrierten Code-Editor aktualisieren. Zur Verwendung dieses Features muss Ihre Funktion folgende Kriterien erfüllen:

- Ihre Funktion muss eine der interpretierten Sprachlaufzeiten verwenden (Python oder Ruby) Node.js
- Das Bereitstellungspaket Ihrer Funktion muss kleiner als 50 MB (entpackt) sein.

Funktionscode für Funktionen mit Container-Image-Bereitstellungspaketen kann nicht direkt in der Konsole bearbeitet werden.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.
4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Befehle Funktion erstellen und Funktionscode aktualisieren, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre ZIP-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip-Datei mit dem zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten erstellt. Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Sie können es auch verwenden AWS SAM , um eine Lambda-Funktion mithilfe eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion zu erstellen AWS SAM, können Sie Ihre ZIP-Datei in einem Amazon S3 S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer AWS SAM Vorlage spezifiziert die `AWS::Serverless::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri`- auf die Amazon S3 S3-URI, den Pfad zum lokalen Ordner oder FunctionCodeObjekt des Funktionscodes gesetzt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Wenn Ihre ZIP-Datei größer als 50 MB ist, müssen Sie sie nicht zuerst in einen Amazon S3 S3-Bucket hochladen. AWS SAM AWS SAM kann .zip-Pakete bis zur maximal zulässigen Größe von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei in finden Sie unter AWS::Serverless: AWS SAM:Function im Entwicklerhandbuch.AWS SAM

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit CloudFormation

Sie können verwenden CloudFormation , um eine Lambda-Funktion mithilfe eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die CloudFormation generierte ZIP-Datei darf 4 MB nicht überschreiten. Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei finden Sie unter AWS::Lambda: :Function im Benutzerhandbuch. CloudFormationCloudFormation

Stellen Sie Node.js Lambda-Ereignisfunktionen mit Container-Images bereit

Es gibt drei Möglichkeiten, ein Container-Image für eine Node.js Lambda-Ereignisfunktion zu erstellen:

- Verwenden eines AWS Basisbild für Node.js

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Mit einem AWS OS-only Basisbild

  AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Runtime-Interface-Client für Node.js in das Image aufnehmen.

- Verwenden Sie ein Image, das nicht zur AWS Basisversion gehört

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Runtime-Interface-Client für Node.js in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für Node.js

- Verwenden eines AWS Basisbild für Node.js
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für Node.js

AWS bietet die folgenden Basis-Images für Node.js:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
26	Node.js 26	Amazon Linux 2023	Dockerfile für 26 Node.js GitHub	Nicht geplant
24	Node.js 24	Amazon Linux 2023	Dockerfile für 24 Node.js GitHub	30. April 2028
22	Node.js 22	Amazon Linux 2023	Dockerfile für 22 auf Node.js GitHub	30. April 2027

Amazon ECR-Repository: gallery.ecr. aws/lambda/node.js

Die Basis-Images Node.js 20 und höher basieren auf dem Minimal-Container-Image von Amazon Linux 2023. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie `glibc`.

AL2023-based Bilder verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, was der Standard-Paketmanager in Amazon Linux 2 ist. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2023-based Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2023 und Amazon Linux 2 finden Sie unter Introducing the Amazon Linux 2023 runtime for AWS Lambda im AWS -Compute-Blog.

> **Note**
>
> Um AL2023-based Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie die Docker-Version 20.10.10 oder höher verwenden.

Verwenden eines AWS Basisbild für Node.js

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Node.js

Erstellen eines Images aus einem Base Image

Um ein Container-Image aus einem zu erstellen AWS Basis-Image für Node.js

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie ein neues Node.js Projekt mit`npm`. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

   ```
   npm init
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `index.js`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example CommonJS-Handler

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
   ```

```
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. Wenn Ihre Funktion von anderen Bibliotheken als den abhängt AWS SDK für JavaScript, verwenden Sie npm, um sie Ihrem Paket hinzuzufügen.
5. Erstellen Sie eine neue Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die FROM-Eigenschaft auf den URI des Basis-Images.
   - Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in eine Lambda-defined Umgebungsvariable zu {LAMBDA_TASK_ROOT} kopieren.
   - Legen Sie das CMD-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

   Example Dockerfile

   ```
   FROM public.ecr.aws/lambda/nodejs:22

   # Copy function code
   COPY index.js ${LAMBDA_TASK_ROOT}

   # Set the CMD to your handler (could also be done as a parameter override outside
    of the Dockerfile)
   CMD [ "index.handler" ]
   ```

6. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

   ```
   docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
    .
   ```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.
   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

- `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
- Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-only Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Runtime-Interface-Client in Ihr Image aufnehmen. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Node.js Runtime-Interface-Client mit dem npm-Paketmanager:

```
npm install aws-lambda-ric
```

Sie können den Node.js Runtime-Interface-Client auch von GitHub herunterladen.

Das folgende Beispiel zeigt, wie ein Container-Image für die Node.js Verwendung eines AWS Nicht-Basis-Images erstellt wird. Das Beispiel-Dockerfile verwendet ein `bookworm`-Basis-Image. Das Docker-File enthält den Laufzeitschnittstellen-Client.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Node.js

Erstellen eines Images aus einem alternativen Basis-Image

Um ein Container-Image aus einem Nicht-AWS Basis-Image

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie ein neues Node.js Projekt mit`npm`. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

   ```
   npm init
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `index.js`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example CommonJS-Handler

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
           body: JSON.stringify('Hello from Lambda!'),
       };
       return response;
   ```

```
};
```

4. Erstellen Sie eine neue Docker-Datei. Das folgende Dockerfile verwendet ein `bookworm`-Basis-Image anstelle eines AWS -Basis-Images. Das Dockerfile enthält den Laufzeitschnittstellen-Client, der das Image mit Lambda kompatibel macht. Das folgende Dockerfile verwendet einen Build mit mehreren Phasen.

   In der ersten Phase wird ein Build-Image erstellt. Dabei handelt es sich um eine Node.js Standardumgebung, in der die Abhängigkeiten der Funktion installiert sind. In der zweiten Phase wird ein schlankeres Image erstellt, das den Funktionscode und seine Abhängigkeiten enthält. Dadurch wird die endgültige Image-Größe reduziert.

   - Legen Sie `FROM`-Eigenschaft auf die Kennung des Basis-Images fest.
   - Verwenden Sie den Befehl `COPY`, um den Funktionscode und die Laufzeitabhängigkeiten zu kopieren.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-bookworm as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
```

```
    make \
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-bookworm-slim

# Required for Node runtimes which use npm@8.6.0+ because
# by default npm writes logs under /home/.npm and Lambda fs is read-only
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
   runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
   download/aws-lambda-rie-arm64
   ```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `/usr/local/bin/npx aws-lambda-ric index.handler` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /usr/local/bin/npx aws-lambda-ric index.handler
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /usr/local/bin/npx aws-lambda-ric index.handler
   ```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.
   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.

7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Node.js Lambda-Ereignisfunktionen

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Fügen Sie die Ebene zu Ihrer Funktion hinzu
- Beispiel-App

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Erstellen Sie den Layer mit derselben Node.js Version, die Sie für die Lambda-Funktion verwenden möchten. Wenn Sie Ihren Layer beispielsweise mit Node.js 24 erstellen, verwenden Sie die Node.js 24-Laufzeit für Ihre Funktion.
- Die ZIP-Datei Ihrer Ebene muss eine der folgenden Verzeichnisstrukturen verwenden:
  - `nodejs/node_modules`
  - `nodejs/node`*X*`/node_modules`(wo *X* ist zum Beispiel Ihre Node.js Version`node22`)

  Weitere Informationen finden Sie unter Ebenenpfade für jede Lambda-Laufzeit.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder Node.js Bibliotheken von Drittanbietern enthalten, die mit npm (z. B. `axios` oder`lodash`) installiert wurden, oder Ihre eigenen JavaScript Module.

Third-party Abhängigkeiten

So erstellen Sie eine Ebene mit npm-Paketen

1. Erstellen Sie die erforderliche Verzeichnisstruktur und installieren Sie Pakete direkt darin:

   ```
   mkdir -p nodejs
   npm install --prefix nodejs lodash axios
   ```

   Mit diesem Befehl werden die Pakete direkt in das Verzeichnis `nodejs/node_modules` installiert. Dies ist die Struktur, die Lambda benötigt.

   > **Note**
   >
   > Stellen Sie bei Paketen mit nativen Abhängigkeiten oder binären Komponenten (wie sharp oder bcrypt) sicher, dass sie mit der Lambda-Linux-Umgebung und der Architektur Ihrer Funktion kompatibel sind. Möglicherweise müssen Sie das Flag `--platform` verwenden:
   >
   > ```
   > npm install --prefix nodejs --platform=linux --arch=x64 sharp
   > ```
   >
   > Bei komplexeren nativen Abhängigkeiten ist es unter Umständen erforderlich, diese in einer Linux-Umgebung zu kompilieren, die der Lambda-Laufzeit entspricht. Zu diesem Zweck können Sie Docker verwenden.

2. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip nodejs/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
   ```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
nodejs/
### package.json
### package-lock.json
### node_modules/
    ### lodash/
    ### axios/
    ### (dependencies of the other packages)
```

> **Note**
>
> - Stellen Sie sicher, dass Ihre ZIP-Datei das `nodejs`-Verzeichnis auf der Stammebene enthält und sich `node_modules` darin befindet. Diese Struktur stellt sicher, dass Lambda Ihre Pakete finden und importieren kann.
> - Die Dateien `package.json` und `package-lock.json` im `nodejs/`-Verzeichnis werden von npm für die Abhängigkeitsverwaltung verwendet, sind aber für die Funktionalität der Ebenen von Lambda nicht erforderlich. Jedes installierte Paket enthält bereits eine eigene `package.json`-Datei, die definiert, wie Lambda das Paket importiert.

Benutzerdefinierte JavaScript Module

So erstellen Sie eine Ebene mithilfe Ihres eigenen Codes

1. Erstellen Sie die erforderliche Verzeichnisstruktur für Ihre Ebene:

   ```
   mkdir -p nodejs/node_modules/validator
   cd nodejs/node_modules/validator
   ```

2. Erstellen Sie eine `package.json`-Datei für Ihr benutzerdefiniertes Modul, um zu definieren, wie es importiert werden soll:

   Example nodejs/node_modules/validator/package.json

   ```
   {
   ```

```
  "name": "validator",
  "version": "1.0.0",
  "type": "module",
  "main": "index.mjs"
}
```

3. Erstellen Sie Ihre JavaScript Moduldatei:

   Example nodejs/node_modules/validator/index.mjs

```
export function validateOrder(orderData) {
  // Validates an order and returns formatted data
  const requiredFields = ['productId', 'quantity'];

  // Check required fields
  const missingFields = requiredFields.filter(field => !(field in orderData));
  if (missingFields.length > 0) {
    throw new Error(`Missing required fields: ${missingFields.join(', ')}`);
  }

  // Validate quantity
  const quantity = orderData.quantity;
  if (!Number.isInteger(quantity) || quantity < 1) {
    throw new Error('Quantity must be a positive integer');
  }

  // Format and return the validated data
  return {
    productId: String(orderData.productId),
    quantity: quantity,
    shippingPriority: orderData.priority || 'standard'
  };
}

export function formatResponse(statusCode, body) {
  // Formats the API response
  return {
    statusCode: statusCode,
    body: JSON.stringify(body)
  };
}
```

4. Verpacken Sie den Ebeneninhalt:

Linux/macOS

```
zip -r layer.zip nodejs/
```

PowerShell

```
Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
nodejs/
### node_modules/
    ### validator/
        ### package.json
        ### index.mjs
```

5. Importieren und verwenden Sie die Module in Ihrer Funktion. Beispiel:

```
import { validateOrder, formatResponse } from 'validator';

export const handler = async (event) => {
  try {
    // Parse the order data from the event body
    const orderData = JSON.parse(event.body || '{}');

    // Validate and format the order
    const validatedOrder = validateOrder(orderData);

    return formatResponse(200, {
      message: 'Order validated successfully',
      order: validatedOrder
    });
  } catch (error) {
    if (error instanceof Error && error.message.includes('Missing required
 fields')) {
      return formatResponse(400, {
        error: error.message
      });
    }
```

```
    return formatResponse(500, {
      error: 'Internal server error'
    });
  }
};
```

Sie können das folgende Testereignis verwenden, um die Funktion aufzurufen:

```
{
    "body": "{\"productId\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Erwartete Antwort:

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"productId\":
\"ABC123\",\"quantity\":2,\"shippingPriority\":\"express\"}}"
}
```

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den AWS CLI Befehl publish-layer-version aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes nodejs24.x
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.

3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.
4. (Optional) Wählen Sie unter Kompatible Laufzeiten die Node.js Laufzeit aus, die der Node.js Version entspricht, mit der Sie Ihren Layer erstellt haben.
5. Wählen Sie Erstellen aus.

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Um den Layer an Ihre Funktion anzuhängen, führen Sie den Befehl AWS CLI update-function-configuration aus. Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

> **Note**
>
> Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Beispiel-App

Weitere Beispiele für die Verwendung von Lambda-Layern finden Sie in der Layer-Nodejs-Beispielanwendung im Developer Guide-Repository. AWS Lambda GitHub Diese Anwendung enthält eine Ebene, die die lodash-Bibliothek enthält. Nach dem Erstellen der Ebene können Sie die entsprechende Funktion bereitstellen und aufrufen, um zu bestätigen, dass die Ebene wie erwartet funktioniert.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Node.js Ereignisfunktion

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Methoden

- `getRemainingTimeInMillis()` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `functionName` – Der Name der Lambda-Funktion.
- `functionVersion` – Die Version der Funktion.
- `invokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `awsRequestId` – Der Bezeichner der Aufrufanforderung.
- `logGroupName` – Protokollgruppe für die Funktion.
- `logStreamName` – Der Protokollstream für die Funktions-Instance.
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
  - `cognitoIdentityId`— Die authentifizierte Amazon-Cognito-Identität.
  - `cognitoIdentityPoolId` – Der Amazon-Cognito-Identitätspool, der den Aufruf autorisiert hat.
- `clientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`

  - `env.make`
  - `env.model`
  - `env.locale`
  - `custom` – Benutzerdefinierte Werte, die durch die mobilen Anwendung festgelegt werden.
- `callbackWaitsForEmptyEventLoop`— Standardmäßig (`true`) wartet Lambda bei Verwendung eines Callback-basierten Funktionshandlers darauf, dass die Ereignisschleife leer ist, nachdem der Callback ausgeführt wurde, bevor der Funktionsaufruf beendet wird. Legt fest, dass `false` die Antwort gesendet und der Aufruf sofort beendet wird, nachdem der Callback ausgeführt wurde, anstatt darauf zu warten, dass die Event-Schleife leer ist. Ausstehende Ereignisse werden beim nächsten Aufruf weiterhin ausgeführt. Beachten Sie, dass Lambda Callback-basierte Funktionshandler nur für Node.js 22 und frühere Laufzeiten unterstützt.

Die folgende Beispielfunktion protokolliert Kontextinformationen und gibt den Speicherort der Protokolle zurück.

Example Datei index.js

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

Node.js Lambda-Ereignisfunktionen protokollieren und überwachen

AWS Lambda überwacht Lambda Event Functions automatisch in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Funktion umfasst eine CloudWatch Logs-Protokollgruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Lambda Event Function-Logs an CloudWatch Logs senden.

Auf dieser Seite wird beschrieben, wie Sie eine Protokollausgabe aus dem Code Ihrer Lambda-Funktion erstellen und mithilfe der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Verwenden von erweiterten Lambda-Protokollierungssteuerelementen mit Node.js
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie Methoden auf dem Konsolenobjekt oder eine Protokollierungsbibliothek verwenden, die zu `stdout` oder `stderr` schreibt. Das folgende Beispiel protokolliert die Werte der Umgebungsvariablen und des Ereignisobjekts.

> **Note**
>
> Wir empfehlen, bei der Protokollierung von Eingaben Techniken wie Eingabevalidierung und Ausgabekodierung zu verwenden. Wenn Sie Eingabedaten direkt protokollieren, kann ein Angreifer Ihren Code verwenden, um Manipulationen schwer zu erkennen, Protokolleinträge zu fälschen oder Protokollüberwachungen zu umgehen. Weitere Informationen finden Sie unter Unsachgemäße Neutralisierung der Ausgabe von Protokollen in der Common Weakness Enumeration.

Example Datei index.js – Protokollierung

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
  return context.logStreamName
}
```

Example Protokollformat

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 296 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

Die Node.js Runtime protokolliert die `REPORT` Zeilen `START`–`END`, und für jeden Aufruf. Sie fügt jedem von der Funktion protokollierten Eintrag einen Zeitstempel, eine Anforderungs-ID und eine Protokollebene hinzu. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Sie können Logs in der Lambda-Konsole, in der CloudWatch Logs-Konsole oder über die Befehlszeile anzeigen.

Verwenden von erweiterten Lambda-Protokollierungssteuerelementen mit Node.js

Um Ihnen mehr Kontrolle darüber zu geben, wie die Protokolle Ihrer Funktionen erfasst, verarbeitet und genutzt werden, können Sie die folgenden Protokollierungsoptionen für unterstützte Laufzeiten konfigurieren: Node.js

- Protokollformat — Wählen Sie zwischen Klartext und strukturiertem JSON-Format für die Logs Ihrer Funktion
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, die Lambda an Amazon sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Protokolle sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.

Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Node.js Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden Sie strukturierte JSON-Logs mit Node.js

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die Protokollausgabe mit den Konsolenmethoden`console.trace`,`console.debug`,, `console.log` `console.info`console.error, und `console.warn` an CloudWatch als strukturiertes JSON. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Abhängig von der Protokollierungsmethode, die Ihre Funktion verwendet, kann dieses JSON-Objekt auch zusätzliche Schlüsselpaare enthalten. Wenn Ihre Funktion beispielsweise `console`-Methoden verwendet, um Fehlerobjekte mit mehreren Argumenten zu protokollieren, enthält das JSON-Objekt zusätzliche Schlüssel-Wert-Paare mit den Schlüsseln `errorMessage`, `errorType` und `stackTrace`.

Wenn Ihr Code bereits eine andere Logging-Bibliothek wie Powertools for verwendet AWS Lambda, um strukturierte JSON-Logs zu erstellen, müssen Sie keine Änderungen vornehmen. Lambda codiert keine Protokolle doppelt, die bereits JSON-kodiert sind, sodass die Anwendungsprotokolle Ihrer Funktion weiterhin wie zuvor erfasst werden.

Weitere Informationen zur Verwendung des Pakets Powertools for AWS Lambda Logging zur Erstellung strukturierter JSON-Logs in der Laufzeit finden Sie unter. Node.js the section called “Protokollierung”

Beispiel für Protokollausgaben im JSON-Format

Die folgenden Beispiele zeigen, wie verschiedene Protokollausgaben, die mithilfe der `console` Methoden mit einzelnen und mehreren Argumenten generiert wurden, in CloudWatch Logs erfasst werden, wenn Sie das Protokollformat Ihrer Funktion auf JSON setzen.

Im ersten Beispiel wird die `console.error`-Methode verwendet, um eine einfache Zeichenfolge auszugeben.

Example Node.js Protokollierungscode

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON-Protokolldatensatz

```
{
    "timestamp":"2025-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

Sie können mit den `console`-Methoden auch komplexere strukturierte Protokollnachrichten ausgeben, indem Sie entweder einzelne oder mehrere Argumente verwenden. Im nächsten Beispiel verwenden Sie `console.log`, um zwei Schlüssel-Wert-Paare mit einem einzigen Argument auszugeben. Beachten Sie, dass das `"message"` Feld im JSON-Objekt, das Lambda an CloudWatch Logs sendet, nicht stringifiziert ist.

Example Node.js Logging-Code

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON-Protokolldatensatz

```
{
```

```
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

Im nächsten Beispiel verwenden Sie erneut die `console.log`-Methode, um eine Protokollausgabe zu erstellen. Diesmal verwendet die Methode zwei Argumente, eine Map mit zwei Schlüssel-Wert-Paaren und eine identifizierende Zeichenfolge. Beachten Sie, dass Lambda in diesem Fall das Feld `"message"` in eine Zeichenfolge umwandelt, da Sie zwei Argumente angegeben haben.

Example Node.js Protokollierungscode

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda weist Ausgaben, die mit `console.log` generiert wurden, der Protokollebene INFO zu.

Das letzte Beispiel zeigt, wie Fehlerobjekte mithilfe der `console` Methoden in CloudWatch Logs ausgegeben werden können. Beachten Sie, dass Lambda die Felder `errorMessage`, `errorType` und `stackTrace` zur Protokollausgabe hinzufügt, wenn Sie Fehlerobjekte mit mehreren Argumenten protokollieren.

Example Node.js Logging-Code

```
export const handler = async (event) => {
```

```
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2025-12-08T23:21:04.646Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

Bei der Protokollierung mehrerer Fehlertypen werden die zusätzlichen Felder `errorMessage`, `errorType` und `stackTrace` aus dem ersten Fehlertyp extrahiert, der an die `console`-Methode übergeben wurde.

Verwenden von Clientbibliotheken im Embedded Metric Format (EMF) mit strukturierten JSON-Protokollen

AWS stellt Open-Source-Clientbibliotheken bereit Node.js , mit denen Sie Logs im Embedded Metric Format (EMF) erstellen können. Wenn Sie bereits über Funktionen verfügen, die diese Bibliotheken verwenden, und Sie das Protokollformat Ihrer Funktion in JSON ändern, werden die von Ihrem Code ausgegebenen Metriken CloudWatch möglicherweise nicht mehr erkannt.

Wenn Ihr Code derzeit EMF-Protokolle direkt mit `console.log` oder mithilfe von Powertools for AWS Lambda (TypeScript) CloudWatch ausgibt, kann er diese auch nicht analysieren, wenn Sie das Protokollformat Ihrer Funktion in JSON ändern.

> **Important**
>
> Um sicherzustellen, dass die EMF-Protokolle Ihrer Funktionen weiterhin ordnungsgemäß analysiert werden, aktualisieren Sie Ihre EMF- und Powertools for CloudWatch -Bibliotheken auf die neuesten Versionen. AWS Lambda Wenn Sie zum JSON-Protokollformat wechseln, empfehlen wir Ihnen zudem, Tests durchzuführen, um die Kompatibilität mit den eingebetteten Metriken Ihrer Funktion sicherzustellen. Wenn Ihr Code EMF-Protokolle direkt mit `console.log` ausgibt, ändern Sie Ihren Code so, dass diese Metriken direkt in `stdout` ausgegeben werden, wie im folgenden Codebeispiel gezeigt.

Example **`Code, der eingebettete Metriken an stdout ausgibt`**

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
```

```
                }]
            }]
        },
        "functionVersion": "$LATEST",
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

Filterung auf Protokollebene verwenden mit Node.js

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mit den Standardmethoden der Konsole und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. AWS Lambda filtert dann Ihre Protokollausgaben mithilfe des Schlüsselwertpaars „Level“ im JSON-Objekt, wie unter beschriebenthe section called “Verwenden Sie strukturierte JSON-Logs mit Node.js”. Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unterthe section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.
- Verwenden Sie eine andere Logging-Bibliothek oder Methode, um strukturierte JSON-Logs in Ihrem Code zu erstellen, die ein „Level“ -Schlüsselwertpaar enthalten, das die Ebene der Protokollausgabe definiert. Sie können Powertools beispielsweise verwenden, AWS Lambda um strukturierte JSON-Protokollausgaben aus Ihrem Code zu generieren. Weitere Informationen the section called “Protokollierung” zur Verwendung von Powertools mit der Runtime finden Sie unter Node.js .

  Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Wenn Sie Ihre Funktion für die Filterung auf Protokollebene konfigurieren, wählen Sie aus den folgenden Optionen die Ebene der Protokolle aus, die Sie an Logs senden AWS Lambda möchten CloudWatch :

Protokollebene	Standardnutzung
TRACE (am detailliertesten)	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Detaillierte Informationen für das System-Debugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Meldungen über potenzielle Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert
FATAL (am wenigsten Details)	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Lambda sendet Protokolle der ausgewählten Stufe und niedriger bis CloudWatch. Wenn Sie beispielsweise die Protokollebene WARN konfigurieren, sendet Lambda Protokolle, die den Stufen WARN, ERROR und FATAL entsprechen.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
```

```
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
```

```
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Node.js Code instrumentieren in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können es verwenden X-Ray, um eine Anfrage zu verfolgen, während sie Ressourcen in Ihrer Anwendung durchquert, zu denen Lambda-Funktionen und andere AWS Dienste gehören können.

Um Ablaufverfolgungsdaten an zu senden X-Ray, können Sie eine von zwei SDK-Bibliotheken verwenden:

- AWS Distribution for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des (OTel) SDK. OpenTelemetry
- AWS X-Ray-SDK für Node.js— Ein SDK zum Generieren und Senden von Trace-Daten an. X-Ray

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray Dienst zu senden. Anschließend können Sie X-Ray die Leistungskennzahlen Ihrer Anwendung anzeigen, filtern und Einblicke in sie gewinnen, um Probleme und Optimierungsmöglichkeiten zu identifizieren.

> **Important**
>
> Die SDKs X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von angeboten wird. AWS Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgängige Ablaufverfolgung implementieren. X-Ray Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Auswahl zwischen der AWS Distribution for Open Telemetry und SDKs. X-Ray

Sections

- Verwenden Sie ADOT zur Instrumentierung Ihrer Funktionen Node.js
- Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Funktionen Node.js
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Die Ablaufverfolgung wird aktiviert mit CloudFormation
- Interpretieren und X-Ray Trace

- Speichern von Laufzeitabhängigkeiten in einer Ebene (X-Ray SDK)

Verwenden Sie ADOT zur Instrumentierung Ihrer Funktionen Node.js

ADOT bietet vollständig verwaltete Lambda-Ebenen, die alles packen, was Sie zum Sammeln von Telemetriedaten mit dem OTel-SDK benötigen. Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Ebenen auch für die benutzerdefinierte Initialisierung von OTel konfigurieren. Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Node.js Laufzeiten können Sie die AWS verwaltete Lambda-Schicht für ADOT Javascript hinzufügen, um Ihre Funktionen automatisch zu instrumentieren. Eine ausführliche Anleitung zum Hinzufügen dieser Ebene finden Sie unter AWS Distro for OpenTelemetry Lambda Support for JavaScript in der ADOT-Dokumentation.

Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Funktionen Node.js

Um Details zu Aufrufen aufzuzeichnen, die Ihre Lambda-Ereignisfunktion an andere Ressourcen in Ihrer Anwendung tätigt, können Sie auch die AWS X-Ray-SDK für Node.js verwenden. Um das SDK zu erhalten, fügen Sie das `aws-xray-sdk-core`-Paket den Abhängigkeiten Ihrer Anwendung hinzu.

Example leer- .json nodejs/package

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
}
```

Um AWS SDK-Clients in Version 3 zu AWS SDK für JavaScript instrumentieren, schließen Sie die Client-Instanz mit der `captureAWSv3Client` Methode ein.

Example leer — nodejs/function /index.js — Auf der Suche nach einem AWS SDK-Client

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());

// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

Die Lambda-Laufzeit legt einige Umgebungsvariablen fest, um das X-Ray SDK zu konfigurieren. Lambda setzt beispielsweise auf, `AWS_XRAY_CONTEXT_MISSING` um `LOG_ERROR` zu vermeiden, dass Laufzeitfehler vom X-Ray SDK ausgelöst werden. Um eine benutzerdefinierte Strategie für fehlenden Kontext festzulegen, überschreiben Sie die Umgebungsvariable in der Funktionskonfiguration so, dass sie keinen Wert aufweist. Dann können Sie die Strategie für fehlenden Kontext programmgesteuert festlegen.

Example Beispiel-Initialisierungscode

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

Weitere Informationen finden Sie unter the section called “Umgebungsvariablen”.

Nachdem Sie die richtigen Abhängigkeiten hinzugefügt und die erforderlichen Codeänderungen vorgenommen haben, aktivieren Sie die Ablaufverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
```

```
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpretieren und X-Ray Trace

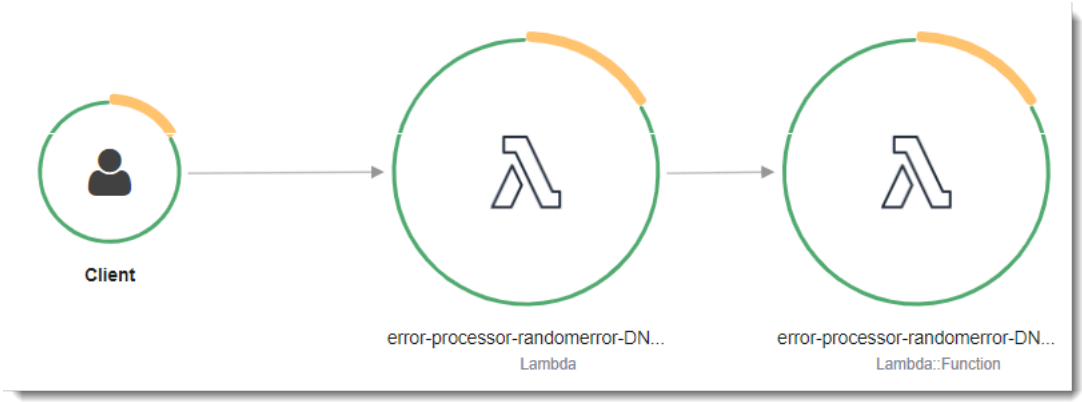
Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen X-Ray. Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray Dienstdiagramm zeigt Informationen zu Ihrer Anwendung und all ihren Komponenten. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK X-Ray, um Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.



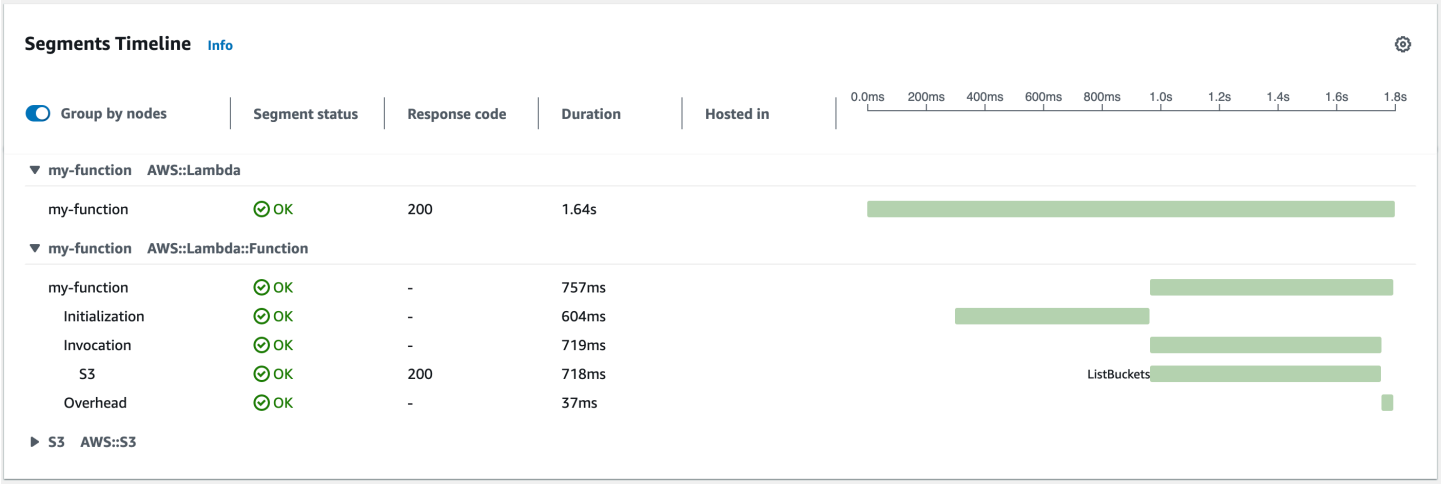
X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

In X-Ray zeichnet ein Trace Informationen über eine Anfrage auf, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem.

Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können Sie geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollnachrichten und Trace-Segmenten feststellen, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.

- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray-SDK für Node.js im AWS X-Ray -Entwicklerhandbuch.

> **Preisgestaltung**
>
> Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit kostenlos nutzen. AWS Bei Überschreitung dieses Schwellenwerts X-Ray fallen Gebühren für die Speicherung und den Abruf von Trace an. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Speichern von Laufzeitabhängigkeiten in einer Ebene (X-Ray SDK)

Wenn Sie das X-Ray SDK verwenden, um AWS SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich umfangreich werden. Um zu vermeiden, dass bei jeder Aktualisierung Ihres Funktionscodes Laufzeitabhängigkeiten hochgeladen werden, packen Sie das X-Ray SDK in eine Lambda-Schicht.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion`-Ressource, die das AWS X-Ray-SDK für Node.js speichert.

Example template.yml — Abhängigkeitsebene

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
```

```
    ...
libs:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: blank-nodejs-lib
    Description: Dependencies for the blank sample app.
    ContentUri: lib/.
    CompatibleRuntimes:
      - nodejs24.x
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Änderungen, um das Layer-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Beispielanwendung blank-nodejs .

Erstellen von Lambda-Eventfunktionen mit TypeScript

Sie können die verwenden Node.js Laufzeit, die ausgeführt werden soll TypeScript einprogrammieren AWS Lambda Event-Funktionen. Weil Node.js läuft nicht TypeScript Code nativ, Sie müssen zuerst Ihren transpilieren TypeScript codieren in JavaScript. Dann benutze den JavaScript Dateien, um Ihren Funktionscode auf Lambda bereitzustellen. Ihr Code läuft in einer Umgebung, die Folgendes umfasst AWS SDK für JavaScript, mit Anmeldeinformationen von einem AWS Identity and Access Management (IAM) -Rolle, die Sie verwalten. Um mehr über die SDK-Versionen zu erfahren, die im Lieferumfang des enthalten sind Node.js Laufzeiten finden Sie unter the section called “Runtime-included SDK-Versionen”.

Für Informationen zur Rückverfolgung TypeScript funktioniert mit, siehe the section called “Nachverfolgung”.

Lambda unterstützt Folgendes Node.js Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 26	`nodejs26.x`	Amazon Linux 2023	Nicht geplant	Nicht geplant	Nicht geplant

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 24	`nodejs24.x`	Amazon Linux 2023	30. April 2028	1. Juni 2028	1. Juli 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	30. April 2027	29. Juli 2027	31. August 2027

Themen

- Einrichtung eines TypeScript Entwicklungsumgebung
- Typdefinitionen für Lambda
- Definieren Sie den Lambda-Ereignisfunktionshandler in TypeScript
- Stellen Sie transpilierten TypeScript Code in Lambda mit ZIP-Dateiarchiven bereit
- Stellen Sie transpilierten TypeScript Code in Lambda mit Container-Images bereit
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur TypeScript Ereignisfunktion
- TypeScript Lambda-Ereignisfunktionen protokollieren und überwachen
- TypeScript Code nachverfolgen in AWS Lambda

Einrichtung eines TypeScript Entwicklungsumgebung

Verwenden Sie eine lokale integrierte Entwicklungsumgebung (IDE) oder einen Texteditor, um Ihre zu schreiben TypeScript Funktionscode. Du kannst nicht erstellen TypeScript Code auf der Lambda-Konsole.

Sie können beides verwenden esbuild oder Microsofts TypeScript Compiler (`tsc`) um deine zu transpilieren TypeScript kodieren in JavaScript. AWS Serverless Application Model (AWS SAM) und AWS Cloud Development Kit (AWS CDK) verwenden beide „esbuild“.

Wenn Sie esbuild verwenden, beachten Sie Folgendes:

- Es gibt mehrere TypeScript Vorbehalte.
- Sie müssen Ihre konfigurieren TypeScript Transpilationseinstellungen, die den entsprechen Node.js Laufzeit, die Sie verwenden möchten. Weitere Informationen finden Sie unter Ziel in der esbuild-

Dokumentation. Für ein Beispiel für eine tsconfig.json Datei, die zeigt, wie man auf ein bestimmtes Ziel abzielt Node.js Version, die von Lambda unterstützt wird, finden Sie in der TypeScript GitHub Endlager.

- esbuild führt keine Typprüfungen durch. Um Typen zu überprüfen, verwenden Sie den `tsc`-Compiler. Führen Sie `tsc -noEmit` aus oder fügen Sie einen `"noEmit"`-Parameter zu Ihrer tsconfig.json-Datei hinzu, wie im folgenden Beispiel gezeigt. Das konfiguriert `tsc` nicht emittieren JavaScript Dateien. Verwenden Sie nach der Überprüfung der Typen esbuild, um die zu konvertieren TypeScript Dateien in JavaScript.

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true,
   "isolatedModules": true,
 },
 "exclude": ["node_modules", "**/*.test.ts"]
}
```

Typdefinitionen für Lambda

Das @types/aws-Lambda Das Paket stellt Typdefinitionen für Lambda-Ereignisfunktionen bereit. Installieren Sie dieses Paket, wenn Ihre Funktion eines der folgenden Elemente verwendet:

- Häufig AWS Ereignisquellen wie:
  - `APIGatewayProxyEvent`: Für Für Proxy-Integrationen in Amazon API Gateway
  - `SNSEvent`: Für Amazon-Simple-Notification-Service-Benachrichtigungen
  - `SQSEvent`: Für Amazon-Simple-Queue-Service-Nachrichten
  - `S3Event`: Für S3-Auslöserereignisse

  - `DynamoDBStreamEvent`: Für Amazon DynamoDB Streams
- Das Lambda-Kontext-Objekt
- Das Callback-Handler-Muster

Um die Lambda-Typdefinitionen zu Ihrer Funktion hinzuzufügen, installieren Sie `@types/aws-lambda` als eine Entwicklungsabhängigkeit:

```
npm install -D @types/aws-lambda
```

Importieren Sie dann die Typen aus `aws-lambda`:

```
import { Context, S3Event, APIGatewayProxyEvent } from 'aws-lambda';

export const handler = async (event: S3Event, context: Context) => {
    // Function code
};
```

Die `import ... from 'aws-lambda'`-Anweisung importiert die Typdefinitionen. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda in der DefinitelyTyped GitHub Endlager.

> **Note**
>
> Du brauchst nicht @types/aws-Lambda wenn Sie Ihre eigenen benutzerdefinierten Typdefinitionen verwenden. Eine Beispielfunktion, die ihren eigenen Typ für ein Ereignisobjekt definiert, finden Sie unter Beispiel für einen TypeScript Lambda-Ereignisfunktionscode.

Definieren Sie den Lambda-Ereignisfunktionshandler in TypeScript

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Ereignisfunktionshandlern arbeiten TypeScript, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine TypeScript Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdateiquittung erstellt und diese Datei in einem Amazon Simple Storage Service (Amazon S3) -Bucket ablegt. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Themen

- Ihr TypeScript Projekt einrichten
- Beispiel für einen TypeScript Lambda-Ereignisfunktionscode
- CommonJS- und ES-Module
- Node.js Initialisierung
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Gültige Handlermuster für Funktionen TypeScript
- Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für TypeScript Lambda-Ereignisfunktionen

Ihr TypeScript Projekt einrichten

Verwenden Sie eine lokale integrierte Entwicklungsumgebung (IDE) oder einen Texteditor, um Ihren TypeScript Funktionscode zu schreiben. Sie können auf der Lambda-Konsole keinen TypeScript Code erstellen.

Es gibt mehrere Möglichkeiten, ein TypeScript Lambda-Projekt zu initialisieren. Sie können beispielsweise ein Projekt mit `npm` erstellen oder eine AWS SAM - bzw. AWS CDK -Anwendung erstellen. So erstellen Sie das Projekt mit `npm`:

```
npm init
```

Ihr Funktionscode befindet sich in einer `.ts` Datei, die Sie bei der Erstellung in eine JavaScript Datei transpilieren. Sie können entweder esbuild oder den TypeScript Compiler (`tsc`) von Microsoft verwenden, um Ihren Code in diese Datei zu transpilieren. TypeScript JavaScript Um „esbuild“ zu verwenden, fügen Sie es als Entwicklungsabhängigkeit hinzu:

```
npm install -D esbuild
```

Ein typisches TypeScript Lambda-Funktionsprojekt folgt dieser allgemeinen Struktur:

```
/project-root
  ### index.ts - Contains main handler
  ### dist/ - Contains compiled JavaScript
  ### package.json - Project metadata and dependencies
  ### package-lock.json - Dependency lock file
  ### tsconfig.json - TypeScript configuration
  ### node_modules/ - Installed dependencies
```

Beispiel für einen TypeScript Lambda-Ereignisfunktionscode

Das folgende Beispiel für einen Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket. Dieses Beispiel definiert einen benutzerdefinierten Ereignistyp (`OrderEvent`). Informationen zum Importieren von Typdefinitionen für AWS Ereignisquellen finden Sie unter Typdefinitionen für Lambda.

> **Note**
>
> In diesem Beispiel wird ein ES-Modul-Handler verwendet. Lambda unterstützt sowohl ES-Modul- als auch CommonJS-Handler. Weitere Informationen finden Sie unter CommonJS- und ES-Module.

Example index.ts Lambda-Ereignisfunktion

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

// Define the shape of the input event
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
export const handler = async (event: OrderEvent): Promise<string> => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
        throw error;
    }
};

/**
```

```
 * Helper function to upload receipt to S3
 */
async function uploadReceiptToS3(bucketName: string, key: string, receiptContent:
 string): Promise<void> {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
    }
}
```

Diese `index.ts`-Datei enthält die folgenden Abschnitte des Codes:

- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt, z. B. AWS SDK-Clients.
- `const s3Client`-Deklaration: Hiermit wird ein Amazon-S3-Client außerhalb der Handler-Funktion initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen und der Client wird für die Wiederverwendung über mehrere Aufrufe hinweg aufbewahrt.
- `type OrderEvent`: Definiert die Struktur des erwarteten Eingabeereignisses.
- `export const handler`: Dies ist die Haupt-Handler-Funktion, die Lambda aufruft. Geben Sie beim Bereitstellen Ihrer Funktion `index.handler` für die Handler-Eigenschaft an. Der Wert der Eigenschaft `Handler` besteht aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt.
- `uploadReceiptToS3`-Funktion: Dies ist eine Hilfsfunktion, auf die von der Haupt-Handler-Funktion verwiesen wird.

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

CommonJS- und ES-Module

Node.js unterstützt zwei Modulsysteme: CommonJS- und ECMAScript-Module (ES-Module). Lambda empfiehlt die Verwendung von ES-Modulen, da sie Await auf oberster Ebene unterstützen, wodurch asynchrone Aufgaben während der Initialisierung der Ausführungsumgebung abgeschlossen werden können.

Node.js behandelt Dateien mit einer `.cjs` Dateinamenerweiterung als CommonJS-Module, während eine `.mjs` Erweiterung ES-Module bezeichnet. Node.js Behandelt Dateien mit der `.js` Dateinamenerweiterung standardmäßig als CommonJS-Module. Sie können so konfigurieren Node.js , dass `.js` Dateien als ES-Module behandelt werden, indem Sie das `type` as `module` in der `package.json` Funktionsdatei angeben. Sie können Node.js in Lambda so konfigurieren, dass automatisch erkannt wird, ob eine `.js` Datei als CommonJS- oder als ES-Modul behandelt werden soll, indem Sie der `NODE_OPTIONS` Umgebungsvariablen das `–experimental-detect-module` Flag hinzufügen. Weitere Informationen finden Sie unter Experimentelle Node.js Funktionen.

Die folgenden Beispiele zeigen Funktionshandler, die sowohl mit ES-Modulen als auch mit CommonJS-Modulen geschrieben wurden. Die übrigen Beispiele auf dieser Seite verwenden alle ES-Module.

Node.js Initialisierung

Node.js verwendet ein nicht blockierendes I/O Modell, das effiziente asynchrone Operationen mithilfe einer Ereignisschleife unterstützt. Wenn Node.js die Funktion beispielsweise einen Netzwerkaufruf tätigt, verarbeitet sie weiterhin andere Operationen, ohne eine Netzwerkantwort zu blockieren. Wenn die Netzwerkantwort empfangen wird, wird sie in die Rückruf-Warteschlange gestellt. Aufgaben aus der Warteschlange werden verarbeitet, wenn die aktuelle Aufgabe abgeschlossen ist.

Lambda empfiehlt die Verwendung von Await auf oberster Ebene, damit asynchrone Aufgaben, die während der Initialisierung der Ausführungsumgebung initiiert wurden, während der Initialisierung abgeschlossen werden. Asynchrone Aufgaben, die während der Initialisierung nicht abgeschlossen werden, werden normalerweise beim ersten Funktionsaufruf ausgeführt. Dies kann zu unerwartetem Verhalten oder Fehlern führen.

Beispielsweise könnte Ihre Funktionsinitialisierung einen Netzwerkaufruf ausführen, um einen Parameter aus dem AWS Parameterspeicher abzurufen. Wenn diese Aufgabe während der Initialisierung nicht abgeschlossen wird, kann der Wert während eines Aufrufs Null sein. Es kann auch zu einer Verzögerung zwischen der Initialisierung und dem Aufruf kommen, was bei zeitkritischen Vorgängen zu Fehlern führen kann. Insbesondere AWS Serviceabrufe können sich auf

zeitkritische Anforderungssignaturen stützen, was dazu führt, dass Serviceabrufe fehlschlagen, wenn der Anruf während der Initialisierungsphase nicht abgeschlossen wird.

Das Abschließen von Aufgaben während der Initialisierung verbessert in der Regel die Kaltstartleistung und die Leistung beim ersten Aufrufen, wenn Provisioned Concurrency verwendet wird. Weitere Informationen finden Sie in unserem Blogbeitrag Using Node.js ES modules and top level await in. AWS Lambda

Namenskonventionen für Handler

Wenn Sie eine Funktion konfigurieren, besteht der Wert der Handler-Einstellung aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt. Der Standardwert für Funktionen, die in der Konsole erstellt werden und für die Beispiele in diesem Handbuch ist `index.handler`. Dies deutet auf die `handler`-Methode hin, die aus der `index.js`- oder `index.mjs`-Datei wurde.

Wenn Sie eine Funktion in der Konsole mit einem anderen Dateinamen oder Funktionshandlernamen erstellen, müssen Sie den Standardhandlernamen bearbeiten.

So ändern Sie den Funktionshandlernamen (Konsole)

1. Öffnen Sie die Seite Functions (Funktionen) der Lambda-Konsole und wählen Sie eine Funktion aus.
2. Wählen Sie die Registerkarte Code (Code).
3. Scrollen Sie nach unten zum Bereich Laufzeiteinstellungen und wählen Sie Bearbeiten.
4. Geben Sie unter Handler den neuen Namen für Ihren Funktionshandler ein.
5. Wählen Sie Speichern.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gängigste und standardmäßigste Eingabeformat für Lambda Event Functions. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in arbeiten TypeScript, können Sie die Form des Eingabeereignisses mithilfe eines Typs oder einer Schnittstelle definieren. In diesem Beispiel definieren wir die Ereignisstruktur mithilfe eines Typs:

```
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}
```

Nachdem Sie den Typ oder die Schnittstelle definiert haben, verwenden Sie ihn bzw. sie in der Signatur Ihres Handlers, um die Typsicherheit zu gewährleisten:

```
export const handler = async (event: OrderEvent): Promise<string> => {
```

TypeScript Überprüft während der Kompilierung, ob das Ereignisobjekt die erforderlichen Felder mit den richtigen Typen enthält. Der TypeScript Compiler meldet beispielsweise einen Fehler, wenn Sie versuchen, eine Zahl oder `event.amount` eine Zeichenfolge zu verwenden`event.order_id`.

Gültige Handlermuster für Funktionen TypeScript

Wir empfehlen, den Funktionshandler async/awaitzur Deklaration zu verwenden, anstatt Callbacks zu verwenden. Async/await ist eine präzise und lesbare Methode, asynchronen Code zu schreiben, ohne dass verschachtelte Callbacks oder verkettete Zusagen erforderlich sind. Mit können Sie Code schreiben async/await, der sich wie synchroner Code liest, aber trotzdem asynchron ist und nicht blockiert.

Für die Beispiele in diesem Abschnitt wird der Typ `S3Event` verwendet. Sie können jedoch alle anderen AWS Ereignistypen im @ types/aws -lambda-Paket verwenden oder Ihren eigenen Ereignistyp definieren. Verwenden von Typen aus @types/aws-lambda:

1. Fügen Sie das Paket @types/aws-lambda als Entwicklungsabhängigkeit hinzu:

   ```
   npm install -D @types/aws-lambda
   ```

2. Importieren Sie die Typen, die Sie benötigen, wie `Context`, `S3Event` oder `Callback`.

Handler für asynchrone Funktionen (empfohlen)

Das `async`-Schlüsselwort kennzeichnet eine Funktion als asynchron, und das `await`-Schlüsselwort unterbricht die Ausführung der Funktion, bis `Promise` aufgelöst ist. Der Handler akzeptiert die folgenden Argumente:

- `event`: Enthält die an Ihre Funktion übergebenen Eingabedaten.
- `context`: Enthält Informationen über den Aufruf, die Funktion und die Ausführungsumgebung. Weitere Informationen finden Sie unter Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur TypeScript Ereignisfunktion.

Hier sind die gültigen Signaturen für das async/await Muster:

```
export const handler = async (event: S3Event): Promise<void> => { };
```

```
export const handler = async (event: S3Event, context: Context): Promise<void> => { };
```

> **Note**
>
> Bei der asynchronen Verarbeitung von Arrays von Elementen sollten Sie „Await“ mit `Promise.all` verwenden, um sicherzustellen, dass alle Vorgänge abgeschlossen werden. Methoden wie `forEach` warten nicht darauf, dass asynchrone Callbacks abgeschlossen werden. Weitere Informationen finden Sie unter Array.prototype.forEach() in der Mozilla-Dokumentation.

Handler für synchrone Funktionen

Wenn Ihre Funktion keine asynchronen Aufgaben ausführt, können Sie einen synchronen Funktionshandler verwenden, der eine der folgenden Funktionssignaturen verwendet:

```
export const handler = (event: S3Event): void => { };
```

```
export const handler = (event: S3Event, context: Context): void => { };
```

Handler für Antwort-Streaming-Funktionen

Lambda unterstützt Response-Streaming mit Node.js. Handler für Antwort-Streaming-Funktionen verwenden den Dekorator AWSLambda.streamifyResponse () und verwenden drei Parameter: event, responseStream und context. Die Funktionssignatur lautet:

```
export const handler = awslambda.streamifyResponse(async (event: APIGatewayProxyEvent,
 responseStream: NodeJS.WritableStream, context: Context) => { });
```

Weitere Informationen finden Sie unter Antwortstreaming für Lambda-Funktionen.

Callback-based Funktionshandler

> **Note**
>
> Callback-based Funktionshandler werden nur bis zu Node.js 22 unterstützt. Ab Node.js 24 sollten asynchrone Aufgaben mithilfe von asynchronen Funktionshandlern implementiert werden.

Callback-based Funktionshandler können die Event-, Kontext- und Callback-Argumente verwenden. Das Callback-Argument erwartet eine Antwort `Error` und diese muss es sein. JSON-serializable

Hier ist die gültige Signatur für das Callback-Handler-Muster:

```
export const handler = (event: S3Event, context: Context, callback: Callback<void>):
 void => { };
```

Die Funktion wird so lange ausgeführt, bis die Ereignisschleife leer ist oder ein Timeout auftritt. Die Antwort wird erst an den Aufrufer gesendet, wenn alle Ereignisschleifenaufgaben abgeschlossen sind. Wenn eine Zeitüberschreitung der Funktion auftritt, wird stattdessen ein Fehler zurückgegeben. Sie können die Laufzeit so konfigurieren, dass die Antwort sofort gesendet wird, indem Sie WaitsForEmptyEventLoopcontext.callback auf false setzen.

Example TypeScript Funktion mit Callback

Das folgende Beispiel verwendet `APIGatewayProxyCallback`, einen Callback-Typ, der speziell für API-Gateway-Integrationen gilt. Die meisten AWS Ereignisquellen verwenden den generischen `Callback` Typ, der in den obigen Signaturen gezeigt wird.

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für JavaScript. Alle unterstützten Node.js Lambda-Laufzeiten enthalten das SDK für JavaScript Version 3. Wir empfehlen jedoch dringend, dass Sie die benötigten AWS SDK-Clients in Ihr Bereitstellungspaket aufnehmen. Dadurch wird die Abwärtskompatibilität bei zukünftigen Lambda-Laufzeit-Updates maximiert.

Um Ihrer Funktion SDK-Abhängigkeiten hinzuzufügen, verwenden Sie den `npm install`-Befehl für die spezifischen SDK-Clients, die Sie benötigen. Im Beispielcode haben wir den Amazon-S3-Client verwendet. Fügen Sie diese Abhängigkeit hinzu, indem Sie den folgenden Befehl in dem Verzeichnis ausführen, das Ihre `package.json`-Datei enthält:

```
npm install @aws-sdk/client-s3
```

Importieren Sie im Funktionscode den Client und die Befehle, die Sie benötigen, wie in der Beispielfunktion gezeigt:

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Initialisieren Sie dann einen Amazon-S3-Client:

```
const s3Client = new S3Client();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn verwenden, um API-Aufrufe für diesen AWS Dienst zu tätigen. Der Beispielcode ruft die Amazon S3 PutObjectS3-API-Aktion wie folgt auf:

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe von `process.env` auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit den folgenden Codezeilen auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb des Handlers. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis ausführt, und der Client bleibt für die Wiederverwendung bei allen Aufrufen verfügbar.

Bewährte Codemethoden für TypeScript Lambda-Ereignisfunktionen

Befolgen Sie diese Richtlinien, wenn Sie Lambda-Funktionen erstellen:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.

- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Node.js und Python-Laufzeiten gehören dazu die AWS SDKs. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen

Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Stellen Sie transpilierten TypeScript Code in Lambda mit ZIP-Dateiarchiven bereit

Bevor Sie TypeScript Code bereitstellen können AWS Lambda, müssen Sie ihn transpilieren in. JavaScript Auf dieser Seite werden drei Möglichkeiten zum Erstellen und Bereitstellen von TypeScript Code für Lambda mit ZIP-Dateiarchiven beschrieben:

- Verwenden von AWS Serverless Application Model ()AWS SAM
- Mit dem AWS Cloud Development Kit (AWS CDK)
- Verwenden von AWS Command Line Interface (AWS CLI) und esbuild

AWS SAM und AWS CDK vereinfachen das Erstellen und Bereitstellen von TypeScript Funktionen. Die AWS SAM -Vorlagenspezifikation bietet eine einfache und saubere Syntax zur Beschreibung der Lambda-Funktionen, APIs, Berechtigungen, Konfigurationen und Ereignisse, aus denen Ihre Serverless-Anwendung besteht. Mit dem AWS CDKkönnen Sie zuverlässige, skalierbare und kostengünstige Anwendungen in der Cloud mit der beachtlichen Ausdruckskraft einer Programmiersprache erstellen. Das AWS CDK richtet sich an mäßig bis sehr erfahrene AWS Benutzer. AWS CDK Sowohl der als auch der AWS SAM verwenden Esbuild, um Code zu transpilieren TypeScript . JavaScript

Verwenden AWS SAM um TypeScript Code für Lambda bereitzustellen

Gehen Sie wie folgt vor, um eine Hello TypeScript World-Beispielanwendung herunterzuladen, zu erstellen und bereitzustellen AWS SAM. Diese Anwendung implementiert ein grundlegendes API-Backend. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, wird die Lambda-Funktion aufgerufen. Die Funktion gibt eine `hello world`-Nachricht zurück.

> **Note**
>
> AWS SAM verwendet Esbuild, um Node.js Lambda-Funktionen aus TypeScript Code zu erstellen. Die Esbuild-Unterstützung befindet sich derzeit in der öffentlichen Vorschau. Während der öffentlichen Vorschau kann es bei der Esbuild-Unterstützung zu abwärtsinkompatiblen Änderungen kommen.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher
- Node.js

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --
runtime nodejs24.x
```

2. (Optional) Die Beispielanwendung enthält Konfigurationen für häufig verwendete Tools wie eslLint für Code-Linting und Jest für Einheitentests. So führen Sie Lint- und Testbefehle aus:

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. Entwickeln Sie die App.

```
cd sam-app
sam build
```

4. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

5. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, antworten Sie mit `Enter`.
6. Die Ausgabe zeigt den Endpunkt für die REST-API. Öffnen Sie den Endpunkt in einem Browser, um die Funktion zu testen. Folgende Antwort sollte angezeigt werden:

```
{"message":"hello world"}
```

7. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwendung der AWS CDK um TypeScript Code für Lambda bereitzustellen

Gehen Sie wie folgt vor, um eine TypeScript Beispielanwendung mit dem zu erstellen und bereitzustellen AWS CDK. Diese Anwendung implementiert ein grundlegendes API-Backend. Es besteht aus einem API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, wird die Lambda-Funktion aufgerufen. Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- AWS CDK Ausführung 2
- Node.js
- Entweder Docker oder esbuild

Stellen Sie ein Beispiel bereit AWS CDK Anwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initialisieren Sie die App.

   ```
   cdk init app --language typescript
   ```

3. Fügen Sie das @ types/aws -Lambda-Paket als Entwicklungsabhängigkeit hinzu. Dieses Paket enthält die Typdefinitionen für Lambda.

   ```
   npm install -D @types/aws-lambda
   ```

4. Öffnen Sie das lib-Verzeichnis. Sie sollten eine Datei mit dem Namen hello-world-stack.ts sehen. Erstellen Sie neue zwei neue Dateien in diesem Verzeichnis: hello-world.function.ts und hello-world.ts.
5. Öffnen Sie hello-world.function.ts und fügen Sie den folgenden Code in die Datei ein. Dies ist der Code für die Lambda-Funktion.

> **Note**
>
> Die `import` Anweisung importiert die Typdefinitionen aus @ types/aws -lambda. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im Repository. DefinitelyTyped GitHub

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. Öffnen Sie hello-world.ts und fügen Sie den folgenden Code in die Datei ein. Dies enthält das NodejsFunction Konstrukt, das die Lambda-Funktion erstellt, und das LambdaRestApi Konstrukt, das die REST-API erstellt.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
}
```

Das `NodejsFunction`-Konstrukt geht standardmäßig von folgendem aus:

- Ihr Funktions-Handler heißt `handler`.
- Die .ts-Datei, die den Funktionscode enthält (hello-world.function.ts), befindet sich im selben Verzeichnis wie die .ts-Datei, die das Konstrukt (hello-world.ts) enthält. Das Konstrukt verwendet die ID des Konstrukts („hello-world“) und den Namen der Lambda-Handler-Datei („function“), um den Funktionscode zu finden. Wenn sich Ihr Funktionscode beispielsweise in einer Datei namens hello-world.my-function.ts befindet, muss die Datei hello-world.ts wie folgt auf den Funktionscode verweisen:

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

Sie können dieses Verhalten ändern und andere esbuild-Parameter konfigurieren. Weitere Informationen finden Sie unter Konfiguration von esbuild in der AWS CDK API-Referenz.

7. Öffnen Sie hello-world-stack.ts. Dies ist der Code, der Ihren AWS CDK -Stack definiert. Ersetzen Sie den Code mit Folgendem:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. Stellen Sie Ihre Anwendung aus dem Verzeichnis `hello-world` bereit, das Ihre `cdk.json` Datei enthält.

```
cdk deploy
```

9. Der AWS CDK erstellt und verpackt die Lambda-Funktion mithilfe von esbuild und stellt die Funktion dann für die Lambda-Laufzeit bereit. Die Ausgabe zeigt den Endpunkt für die REST-API. Öffnen Sie den Endpunkt in einem Browser, um die Funktion zu testen. Folgende Antwort sollte angezeigt werden:

```
{"message":"hello world"}
```

Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

Verwendung der AWS CLI und esbuild, um TypeScript Code für Lambda bereitzustellen

Das folgende Beispiel zeigt, wie TypeScript Code mithilfe von esbuild transpiliert und für Lambda bereitgestellt wird AWS CLI. esbuild erzeugt eine JavaScript Datei mit allen Abhängigkeiten. Dies ist die einzige Datei, die Sie dem ZIP-Archiv hinzufügen müssen.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Node.js
- Eine Ausführungsrolle für die Lambda-Funktion
- Für Windows-Benutzer ein ZIP-Datei-Hilfsprogramm wie 7zip.

Bereitstellen einer Beispielfunktion

1. Erstellen Sie auf Ihrem lokalen Computer ein Projektverzeichnis für Ihre neue Funktion.
2. Erstellen Sie ein neues Node.js Projekt mit npm oder einem Paketmanager Ihrer Wahl.

   ```
   npm init
   ```

3. Fügen Sie die Pakete @ types/aws -lambda und esbuild als Entwicklungsabhängigkeiten hinzu. Das `@types/aws-lambda`-Paket enthält die Typdefinitionen für Lambda.

   ```
   npm install -D @types/aws-lambda esbuild
   ```

4. Erstellen Sie eine neue Datei mit dem Namen index.ts. Fügen Sie den folgenden Code zur Datei hinzu. Dies ist der Code für die Lambda-Funktion. Die Funktion gibt eine `hello world`-Nachricht zurück. Die Funktion erstellt keine API-Gateway-Ressourcen.

> **Note**
>
> Die `import` Anweisung importiert die Typdefinitionen aus @ types/aws -lambda. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im Repository. DefinitelyTyped GitHub

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. Hinzufügen eines Entwicklungs-Skripts zur package.json-Datei. Dies konfiguriert esbuild so, dass das ZIP-Bereitstellungspaket automatisch erstellt wird. Weitere Informationen finden Sie unter Entwicklungs-Skripte in der esbuild-Dokumentation.

   Linux and MacOS

   ```
   "scripts": {
     "prebuild": "rm -rf dist",
     "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
   target=es2020 --outfile=dist/index.js",
     "postbuild": "cd dist && zip -r index.zip index.js*"
   },
   ```

Windows

In diesem Beispiel verwendet der `"postbuild"` Befehl das 7zip-Hilfsprogramm, um Ihre ZIP-Datei zu erstellen. Verwenden Sie Ihr bevorzugtes Windows-ZIP-Hilfsprogramm und ändern Sie den Befehl nach Bedarf.

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. Erstellen Sie das Paket.

```
npm run build
```

7. Erstellen Sie eine Lambda-Funktion mit dem ZIP-Bereitstellungspaket. Ersetzen Sie den markierten Text durch den Amazon-Ressourcennamen (ARN) Ihrer Ausführungsrolle.

```
aws lambda create-function --function-name hello-world --runtime "nodejs24.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. Führen Sie ein Testereignis aus, um zu bestätigen, dass die Funktion die folgende Antwort zurückgibt. Wenn Sie diese Funktion mit API Gateway aufrufen möchten, Erstellen und Konfigurieren Sie eine REST-API.

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

Stellen Sie transpilierten TypeScript Code in Lambda mit Container-Images bereit

Sie können Ihren TypeScript Code als Node.js Container-Image für eine AWS Lambda Event-Funktion bereitstellen. AWS stellt Basis-Images bereit Node.js , die Ihnen bei der Erstellung des Container-Images helfen. Diese Basis-Images sind mit einer Sprachlaufzeit und anderen Komponenten, die für die Ausführung des Images auf Lambda erforderlich sind, vorinstalliert. AWS stellt für jedes der Basis-Images ein Dockerfile bereit, das Sie beim Erstellen Ihres Container-Images unterstützt.

Wenn Sie ein Community- oder Private-Enterprise-Basis-Image verwenden, müssen Sie den Node.js Runtime Interface Client (RIC) zum Basis-Image hinzufügen, damit es mit Lambda kompatibel ist.

Lambda bietet einen Emulator für die Laufzeitschnittstelle zum lokalen Testen. Die AWS Basis-Images für Node.js enthalten den Runtime-Interface-Emulator. Wenn Sie ein alternatives Basis-Image verwenden (beispielsweise ein Alpine Linux-Image oder ein Debian-Image), können Sie den Emulator in Ihr Image integrieren oder ihn auf Ihrem lokalen Computer installieren.

Verwenden eines Node.js Basis-Images zum Erstellen und Verpacken von TypeScript Funktionscode

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Node.js 2.x

Erstellen eines Images aus einem Base Image

Um ein Bild aus einem zu erstellen AWS Basisbild für Lambda

1. Erstellen Sie auf Ihrem lokalen Computer ein Projektverzeichnis für Ihre neue Funktion.
2. Erstellen Sie ein neues Node.js Projekt mit `npm` oder einem Paketmanager Ihrer Wahl.

   ```
   npm init
   ```

3. Fügen Sie die Pakete @ types/aws -lambda und esbuild als Entwicklungsabhängigkeiten hinzu. Das `@types/aws-lambda`-Paket enthält die Typdefinitionen für Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Hinzufügen eines Entwicklungs-Skripts zur `package.json`-Datei.

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. Erstellen Sie eine neue Datei mit dem Namen `index.ts`. Fügen Sie den folgenden Beispiel-Code zur neuen Datei hinzu. Dies ist der Code für die Lambda-Funktion. Die Funktion gibt eine `hello world`-Nachricht zurück.

> **Note**
>
> Die `import` Anweisung importiert die Typdefinitionen aus @ -lambda. types/aws
> Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im Repository. DefinitelyTyped GitHub

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. Erstellen Sie eine neue Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Legen Sie das `CMD`-Argument zur Angabe des Lambda-Funktionshandlers fest.

Die folgende Beispiel-Docker-Datei verwendet eine mehrstufige Entwicklung. Der erste Schritt transpiliert den Code in. TypeScript JavaScript Im zweiten Schritt wird ein Container-Image erstellt, das nur JavaScript Dateien und Produktionsabhängigkeiten enthält.

Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:22 as builder
WORKDIR /usr/app
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:22
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den

Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.
   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur TypeScript Ereignisfunktion

Wenn Lambda Ihre Event-Funktion ausführt, übergibt es ein Kontextobjekt an den Handler. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Um die Typüberprüfung für das Kontextobjekt zu aktivieren, müssen Sie das @ types/aws -lambda-Paket als Entwicklungsabhängigkeit hinzufügen und den Typ importieren. `Context` Weitere Informationen finden Sie unter Typdefinitionen für Lambda.

Context-Methoden

- `getRemainingTimeInMillis()` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `functionName` – Der Name der Lambda-Funktion.
- `functionVersion` – Die Version der Funktion.
- `invokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `awsRequestId` – Der Bezeichner der Aufrufanforderung.
- `logGroupName` – Protokollgruppe für die Funktion.
- `logStreamName` – Der Protokollstream für die Funktions-Instance.
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
  - `cognitoIdentityId`— Die authentifizierte Amazon-Cognito-Identität.
  - `cognitoIdentityPoolId` – Der Amazon-Cognito-Identitätspool, der den Aufruf autorisiert hat.
- `clientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`

  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`
  - `Custom` – Benutzerdefinierte Werte, die durch die mobilen Anwendung festgelegt werden.
- `callbackWaitsForEmptyEventLoop`— Standardmäßig (`true`) wartet Lambda bei Verwendung eines Callback-basierten Funktionshandlers darauf, dass die Ereignisschleife leer ist, nachdem der Callback ausgeführt wurde, bevor der Funktionsaufruf beendet wird. Legt fest, dass `false` die Antwort gesendet und der Aufruf sofort beendet wird, nachdem der Callback ausgeführt wurde, anstatt darauf zu warten, dass die Event-Schleife leer ist. Ausstehende Ereignisse werden beim nächsten Aufruf weiterhin ausgeführt. Beachten Sie, dass Lambda Callback-basierte Funktionshandler nur für Node.js 22 und frühere Laufzeiten unterstützt.

Example Datei index.js

Die folgende Beispielfunktion protokolliert Kontextinformationen und gibt den Speicherort der Protokolle zurück.

> **Note**
>
> Bevor Sie diesen Code in einer Lambda-Funktion verwenden, müssen Sie das @ types/aws -lambda-Paket als Entwicklungsabhängigkeit hinzufügen. Dieses Paket enthält die Typdefinitionen für Lambda. Weitere Informationen finden Sie unter Typdefinitionen für Lambda.

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

TypeScript Lambda-Ereignisfunktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterLambda Event Function-Logs an CloudWatch Logs senden.

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie Methoden auf dem Konsolenobjekt verwenden. Für eine detailliertere Protokollierung können Sie jede Protokollierungsbibliothek verwenden, die in `stdout` oder `stderr` schreibt.

Sections

- Tools und Bibliotheken für die Protokollierung verwenden
- Verwendung von Powertools für AWS Lambda (TypeScript) und AWS SAM für strukturiertes Logging
- Verwenden von Powertools für AWS Lambda (TypeScript) und die AWS CDK für strukturiertes Logging
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch

Tools und Bibliotheken für die Protokollierung verwenden

Powertools for AWS Lambda (TypeScript) ist ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit. Das Logger-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen über den Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)

- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu
- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (TypeScript) und AWS SAM für strukturiertes Logging

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js 20 oder später
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

Das Ergebnis sieht folgendermaßen aus:

```
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.594000
 2025-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
```

```
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.595000
 2025-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2025-08-31T09:33:10.594Z"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.655000
 2025-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Verwenden von Powertools für AWS Lambda (TypeScript) und die AWS CDK für strukturiertes Logging

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS CDK Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js 20 oder später
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS CDK Anwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initialisieren Sie die App.

   ```
   cdk init app --language typescript
   ```

3. Fügen Sie das @ types/aws -Lambda-Paket als Entwicklungsabhängigkeit hinzu.

   ```
   npm install -D @types/aws-lambda
   ```

4. Installieren Sie das Logger-Dienstprogramm von Powertools.

```
npm install @aws-lambda-powertools/logger
```

5. Öffnen Sie das lib-Verzeichnis. Sie sollten eine Datei mit dem Namen hello-world-stack.ts sehen. Erstellen Sie neue zwei neue Dateien in diesem Verzeichnis: hello-world.function.ts und hello-world.ts.
6. Öffnen Sie hello-world.function.ts und fügen Sie den folgenden Code in die Datei ein. Dies ist der Code für die Lambda-Funktion.

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. Öffnen Sie hello-world.ts und fügen Sie den folgenden Code in die Datei ein. Dies enthält das NodejsFunction Konstrukt, das die Lambda-Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt. Es beinhaltet auch das LambdaRestApi Konstrukt, das die REST-API erstellt.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
```

```
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. Öffnen Sie hello-world-stack.ts. Dies ist der Code, der Ihren AWS CDK -Stack definiert. Ersetzen Sie den Code mit Folgendem:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. Kehren Sie zum Projektverzeichnis zurück.

```
cd hello-world
```

10. Stellen Sie die Anwendung bereit.

```
cdk deploy
```

11. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

13. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name HelloWorldStack
```

Das Ergebnis sieht folgendermaßen aus:

```
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2025-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Logs für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

TypeScript Code nachverfolgen in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können es verwenden X-Ray, um eine Anfrage zu verfolgen, während sie Ressourcen in Ihrer Anwendung durchquert, zu denen Lambda-Funktionen und andere AWS Dienste gehören können.

Um Ablaufverfolgungsdaten an zu senden X-Ray, können Sie eine von drei SDK-Bibliotheken verwenden:

- AWS Distribution for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des (OTel) SDK. OpenTelemetry
- AWS X-Ray SDK für Node.js — Ein SDK zum Generieren und Senden von Trace-Daten an. X-Ray
- Powertools for AWS Lambda (TypeScript) — Ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den Dienst zu senden. X-Ray Anschließend können Sie X-Ray die Leistungskennzahlen Ihrer Anwendung anzeigen, filtern und Einblicke in sie gewinnen, um Probleme und Optimierungsmöglichkeiten zu identifizieren.

> **Important**
>
> Die SDKs X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von angeboten wird. AWS Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgängige Ablaufverfolgung implementieren. X-Ray Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Auswahl zwischen der AWS Distribution for Open Telemetry und SDKs. X-Ray

Sections

- Verwenden von Powertools für AWS Lambda (TypeScript) und AWS SAM zur Rückverfolgung
- Verwenden Sie Powertools für AWS Lambda (TypeScript) und die AWS CDK zur Rückverfolgung
- Interpretation einer X-Ray Ablaufverfolgung

Verwenden von Powertools für AWS Lambda (TypeScript) und AWS SAM zur Rückverfolgung

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

Verwenden Sie Powertools für AWS Lambda (TypeScript) und die AWS CDK zur Rückverfolgung

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS CDK Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS Cloud Development Kit (AWS CDK) Anwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initialisieren Sie die App.

```
cdk init app --language typescript
```

3. Fügen Sie das @ types/aws -Lambda-Paket als Entwicklungsabhängigkeit hinzu.

```
npm install -D @types/aws-lambda
```

4. Installieren Sie das Tracer-Dienstprogramm von Powertools.

```
npm install @aws-lambda-powertools/tracer
```

5. Öffnen Sie das lib-Verzeichnis. Sie sollten eine Datei mit dem Namen hello-world-stack.ts sehen. Erstellen Sie neue zwei neue Dateien in diesem Verzeichnis: hello-world.function.ts und hello-world.ts.
6. Öffnen Sie hello-world.function.ts und fügen Sie den folgenden Code in die Datei ein. Dies ist der Code für die Lambda-Funktion.

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
```

```
      message: 'hello world',
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. Öffnen Sie hello-world.ts und fügen Sie den folgenden Code in die Datei ein. Dies enthält das NodejsFunction Konstrukt, das die Lambda-Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt. Es beinhaltet auch das LambdaRestApi Konstrukt, das die REST-API erstellt.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. Öffnen Sie hello-world-stack.ts. Dies ist der Code, der Ihren AWS CDK -Stack definiert. Ersetzen Sie den Code mit Folgendem:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. Stellen Sie die Anwendung bereit.

```
cd ..
cdk deploy
```

10. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

12. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
```

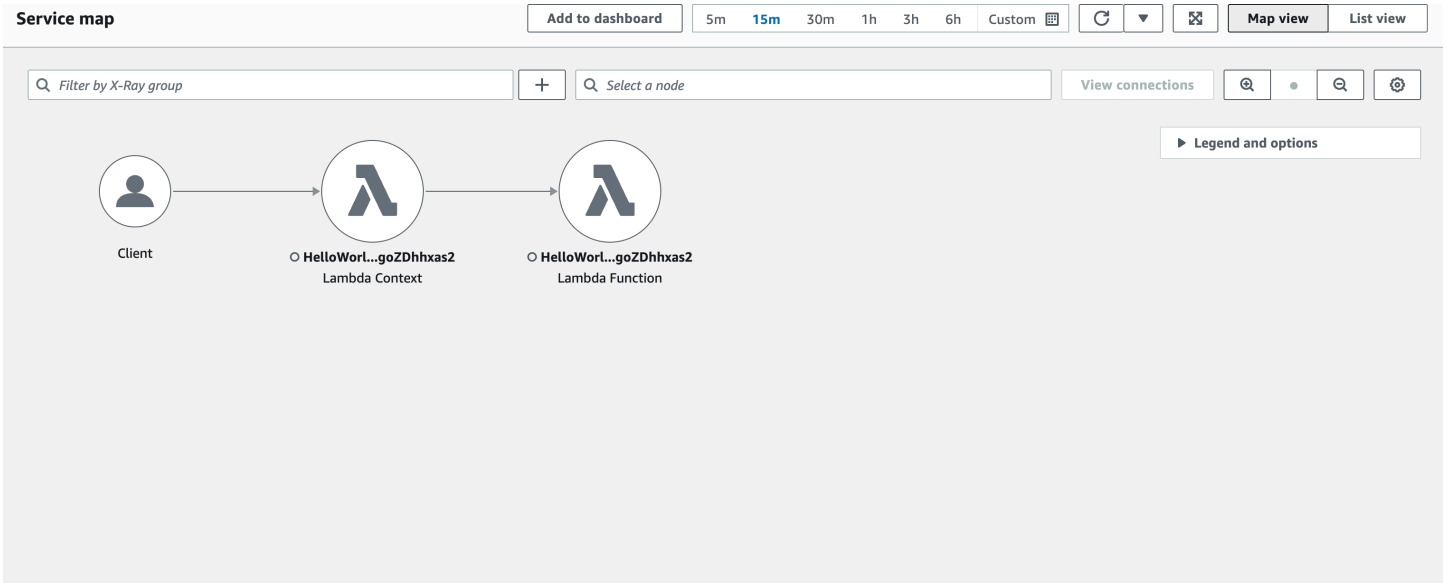
```
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Interpretation einer X-Ray Ablaufverfolgung

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Die X-Ray Trace-Map enthält Informationen über Ihre Anwendung und all ihre Komponenten. Das folgende Beispiel zeigt eine Nachverfolgung aus der Beispielanwendung:



Lambda-Ereignisfunktionen mit Python erstellen

Sie können Python-Code in AWS Lambda Event Functions ausführen. Lambda bietet Laufzeiten für Python, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Umgebung ausgeführt, die das SDK for Python (Boto3) enthält, mit Anmeldeinformationen von einer AWS Identity and Access Management (IAM-) Rolle, die Sie verwalten. Weitere Informationen zu den SDK-Versionen, die in den Laufzeiten von Python enthalten sind, finden Sie unter the section called “Runtime-included SDK-Versionen”.

Lambda unterstützt die folgenden Python-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 3.15	`python3.15`	Amazon Linux 2023	Nicht geplant	Nicht geplant	Nicht geplant
Python 3.14	`python3.14`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.13	`python3.13`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.12	`python3.12`	Amazon Linux 2023	31. Oktober 2028	30. November 2028	10. Januar 2029
Python 3.11	`python3.11`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027
Python 3.10	`python3.10`	Amazon Linux 2	31. Oktober 2026	29. Juli 2027	31. August 2027

So erstellen Sie eine Python-Event-Funktion

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie „Ereignisfunktion erstellen“.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Runtime: Wähle Python 3.14.
4. Wählen Sie „Ereignisfunktion erstellen“.

Die Konsole erstellt eine Lambda-Ereignisfunktion mit einer einzigen Quelldatei namens`lambda_function`. Mit dem integrierten Code-Editor können Sie diese Datei bearbeiten und weitere Dateien hinzufügen. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen

aus, um den Code Ihrer Funktion zu aktualisieren. Um Ihren Code auszuführen, wählen Sie anschließend im Abschnitt TESTEREIGNISSE die Option Testereignis erstellen aus.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Anleitungen zum Testen Ihrer Python-Funktionen finden Sie unterthe section called “Testen”.

Themen

- Runtime-included SDK-Versionen
- Deaktivierte Python-Funktionen
- Reaktionsformat
- Ordnungsgemäßes Herunterfahren von Erweiterungen
- Definieren Sie den Lambda-Ereignisfunktionshandler in Python
- Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions
- Stellen Sie Python-Lambda-Ereignisfunktionen mit Container-Images bereit
- Arbeiten mit Ebenen für Python-Lambda-Ereignisfunktionen
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Python-Ereignisfunktion
- Python-Lambda-Ereignisfunktionen protokollieren und überwachen
- AWS Lambda Testen von Event-Funktionen in Python
- Instrumentierung von Python-Code in AWS Lambda

Runtime-included SDK-Versionen

Die Version des AWS SDK, die in der Python-Laufzeit enthalten ist, hängt von der Laufzeitversion und Ihrer ab AWS-Region. Um die Version des SDK zu finden, die in der von Ihnen verwendeten Runtime enthalten ist, erstellen Sie eine Funktion mit dem folgenden Code.

```
import boto3
import botocore

def lambda_handler(event, context):
```

```
print(f'boto3 version: {boto3.__version__}')
print(f'botocore version: {botocore.__version__}')
```

Deaktivierte Python-Funktionen

In der folgenden Tabelle sind Python-Funktionen aufgeführt, die in den von Lambda verwalteten Laufzeiten und Container-Basis-Images für Python deaktiviert sind. Diese Funktionen müssen aktiviert werden, wenn die ausführbare Python-Runtime-Datei kompiliert wird. Sie können nicht mithilfe eines Ausführungszeitflags aktiviert werden. Um diese Funktionen in Lambda zu verwenden, können Sie Ihren eigenen Python-Runtime-Build mit aktivierten Funktionen bereitstellen, indem Sie ein Container-Image oder eine benutzerdefinierte Laufzeit verwenden.

Python-Funktion	Betroffene Python-Versionen	Status
Just-in-Time (JIT) - Compiler	Python 3.13 und höher	Der JIT-Compiler ist experimentell und wird für Produktionsworkloads nicht empfohlen. Es ist daher in den Lambda-Python-Laufzeiten deaktiviert.
Free-threading	Python 3.13 und höher	Kostenloses Threading (Option zum Deaktivieren der globalen Interpretersperre) ist in Lambda-Python-Builds deaktiviert, da sich dies auf die Leistung von Single-Thread-Code auswirkt.

Reaktionsformat

In Python 3.12 und späteren Python-Laufzeiten geben Funktionen Unicode-Zeichen als Teil ihrer JSON-Antwort zurück. Frühere Python-Laufzeiten geben Escape-Sequenzen für Unicode-Zeichen in Antworten zurück. Wenn Sie beispielsweise in Python 3.11 eine Unicode-Zeichenfolge wie "こんにちは" zurückgeben, werden die Unicode-Zeichen mit Ecape-Zeichen versehen und es wird "\u3053\u3093\u306b\u3061\u306f" zurückgegeben. Die Python-3.12-Laufzeit gibt das Original "こんにちは" zurück.

Durch die Verwendung von Unicode-Antworten wird die Größe von Lambda-Antworten reduziert, sodass größere Antworten einfacher in die maximale Nutzlastgröße von 6 MB für synchrone

Funktionen passen können. Im vorherigen Beispiel hat die Escaped-Version 32 Byte – die Unicode-Zeichenfolge hat 17 Byte.

Wenn Sie auf Python 3.12 oder spätere Python-Laufzeiten aktualisieren, müssen Sie möglicherweise Ihren Code anpassen, um das neue Antwortformat zu berücksichtigen. Wenn der Aufrufer Unicode in Escape-Zeichen erwartet, müssen Sie entweder der zurückgebenden Funktion Code hinzufügen, um den Unicode manuell mit Escape-Zeichen zu versehen, oder den Aufrufer so anpassen, dass er die Unicode-Rückgabe verarbeitet.

Ordnungsgemäßes Herunterfahren von Erweiterungen

Python 3.12 und neuere Python-Laufzeiten bieten verbesserte Funktionen zum ordnungsgemäßen Herunterfahren von Funktionen mit externen Erweiterungen. Wenn Lambda eine Ausführungsumgebung beendet, sendet es ein `SIGTERM`-Signal an die Laufzeitumgebung und dann ein `SHUTDOWN`-Ereignis an jede registrierte externe Erweiterung. Sie können das `SIGTERM` Signal in Ihrer Funktion abfangen und Ressourcen wie Datenbankverbindungen, die von der Funktion erstellt wurden, bereinigen.

Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter Verständnis des Lebenszyklus der Lambda-Ausführungsumgebung. Beispiele für die Verwendung von Graceful Shutdown mit Erweiterungen finden Sie im AWS GitHub Samples-Repository.

Definieren Sie den Lambda-Ereignisfunktionshandler in Python

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktionshandlern in Python arbeiten, einschließlich Namenskonventionen, gültigen Handler-Signaturen und Best Practices für Code. Diese Seite enthält auch ein Beispiel für eine Python-Lambda-Ereignisfunktion, die Informationen über eine Bestellung aufnimmt, eine Textdateiquittung erstellt und diese Datei in einem Amazon Simple Storage Service (Amazon S3) -Bucket ablegt.

Themen

- Beispiel für einen Python-Lambda-Ereignisfunktionscode
- Namenskonventionen für Handler
- Verwenden des Lambda-Ereignisobjekts
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Gültige Handler-Signaturen für Python-Handler
- Rückgabe eines Wertes
- Verwendung der AWS SDK für Python (Boto3) in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Bewährte Codemethoden für Python-Lambda-Ereignisfunktionen

Beispiel für einen Python-Lambda-Ereignisfunktionscode

Das folgende Beispiel für einen Python-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket:

Example Python-Lambda-Ereignisfunktion

```
import json
import os
import logging
import boto3

# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

```
# Initialize the logger
logger = logging.getLogger()
logger.setLevel("INFO")

def upload_receipt_to_s3(bucket_name, key, receipt_content):
    """Helper function to upload receipt to S3"""

    try:
        s3_client.put_object(
            Bucket=bucket_name,
            Key=key,
            Body=receipt_content
        )
    except Exception as e:
        logger.error(f"Failed to upload receipt to S3: {str(e)}")
        raise

def lambda_handler(event, context):
    """
    Main Lambda handler function
    Parameters:
        event: Dict containing the Lambda function event data
        context: Lambda runtime context
    Returns:
        Dict containing status message
    """
    try:
        # Parse the input event
        order_id = event['Order_id']
        amount = event['Amount']
        item = event['Item']

        # Access environment variables
        bucket_name = os.environ.get('RECEIPT_BUCKET')
        if not bucket_name:
            raise ValueError("Missing required environment variable RECEIPT_BUCKET")

        # Create the receipt content and key destination
        receipt_content = (
            f"OrderID: {order_id}\n"
            f"Amount: ${amount}\n"
            f"Item: {item}"
        )
```

```
        key = f"receipts/{order_id}.txt"

        # Upload the receipt to S3
        upload_receipt_to_s3(bucket_name, key, receipt_content)

        logger.info(f"Successfully processed order {order_id} and stored receipt in S3
bucket {bucket_name}")

        return {
            "statusCode": 200,
            "message": "Receipt processed successfully"
        }

    except Exception as e:
        logger.error(f"Error processing order: {str(e)}")
        raise
```

Diese -Datei enthält die folgenden Abschnitte des Codes:

- `import`Block: Verwenden Sie diesen Block, um Bibliotheken einzubeziehen, die Ihre Funktion benötigt.
- Globale Initialisierung von SDK-Client und Logger: Durch das Einfügen von Initialisierungscode außerhalb des Handlers wird die Wiederverwendung der Ausführungsumgebung genutzt, um die Leistung Ihrer Funktion zu verbessern. Weitere Informationen hierzu finden Sie unter the section called “Bewährte Codemethoden für Python-Lambda-Ereignisfunktionen”.
- `def upload_receipt_to_s3(bucket_name, key, receipt_content):`: Dies ist eine Hilfsfunktion, die von der `lambda_handler`-Hauptfunktion aufgerufen wird.
- `def lambda_handler(event, context):`: Dies ist die Haupthandler-Funktion für Ihren Code, die Ihre Hauptanwendungslogik enthält. Wenn Lambda Ihren Funktionshandler aufruft, übergibt die Lambda-Laufzeit zwei Argumente an die Funktion: das Ereignisobjekt, das Daten für die Verarbeitung durch Ihre Funktion enthält, und das Kontextobjekt, das Informationen über den Funktionsaufruf enthält.

Namenskonventionen für Handler

Der Name des Funktionshandlers, der bei der Erstellung einer Funktion definiert wurde, ist abgeleitet von:

- dem Namen der Datei, in der sich die Lambda-Handler-Funktion befindet.

- dem Namen der Python-Handler-Funktion.

Wenn die Datei im obigen Beispiel `lambda_function.py` heißt, wird der Handler als `lambda_function.lambda_handler` angegeben. Dies ist der Standard-Handlername für Funktionen, die Sie mit der Lambda-Konsole erstellen.

Wenn Sie eine Funktion in der Konsole mit einem anderen Dateinamen oder Funktionshandlernamen erstellen, müssen Sie den Standardhandlernamen bearbeiten.

So ändern Sie den Funktionshandlernamen (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code (Code).
3. Scrollen Sie nach unten zum Bereich Laufzeiteinstellungen und wählen Sie Bearbeiten.
4. Geben Sie unter Handler den neuen Namen für Ihren Funktionshandler ein.
5. Wählen Sie Speichern.

Verwenden des Lambda-Ereignisobjekts

Wenn Lambda Ihre Funktion aufruft, wird dem Funktionshandler ein Ereignisobjekt-Argument übergeben. JSON-Objekte sind das gängigste Ereignisformat für Lambda-Ereignisfunktionen. Im Codebeispiel im vorherigen Abschnitt erwartet die Funktion eine Eingabe im folgenden Format:

```
{
    "Order_id": "12345",
    "Amount": 199.99,
    "Item": "Wireless Headphones"
}
```

Wenn Ihre Funktion von einer anderen aufgerufen wird AWS-Service, ist das Eingabeereignis ebenfalls ein JSON-Objekt. Das genaue Format des Ereignisobjekts hängt vom Service ab, der Ihre Funktion aufruft. Das Ereignisformat für einen bestimmten Service finden Sie auf der entsprechenden Seite im Kapitel „the section called “Integration anderer Services”“.

Wenn das Eingabeereignis in Form eines JSON-Objekts vorliegt, konvertiert die Lambda-Laufzeit das Objekt in ein Python-Wörterbuch. Um Werte in der Eingabe-JSON Variablen in Ihrem Code zuzuweisen, verwenden Sie die Standardmethoden für Python-Wörterbücher, wie im Beispielcode dargestellt.

Sie können Daten auch als JSON-Array oder als einen der anderen gültigen JSON-Datentypen an Ihre Funktion übergeben. In der folgenden Tabelle ist definiert, wie die Python-Laufzeit diese JSON-Typen konvertiert.

JSON-Datentyp	Python-Datentyp
object	Wörterbuch (`dict`)
Array	Liste (`list`)
number	Ganzzahl (`int`) oder Fließkommazahl (`float`)
Zeichenfolge	Zeichenfolge (`str`)
Boolesch	Boolean () `bool`
Null	NoneType (`NoneType`)

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über den Funktionsaufruf und die Ausführungsumgebung. Lambda übergibt das Kontextobjekt automatisch an Ihre Funktion, wenn diese aufgerufen wird. Mit dem Kontextobjekt können Sie zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion ausgeben.

Das Kontextobjekt ist eine Python-Klasse, die im Schnittstellenclient der Lambda-Laufzeit definiert ist. Um den Wert einer der Eigenschaften des Kontextobjekts zurückzugeben, verwenden Sie die entsprechende Methode des Kontextobjekts. Im folgenden Codeausschnitt wird beispielsweise der Wert der `aws_request_id`-Eigenschaft (die Kennung für die Aufrufanforderung) einer Variablen namens `request` zugewiesen.

```
request = context.aws_request_id
```

Weitere Informationen zur Verwendung des Lambda-Kontextobjekts und eine vollständige Liste der verfügbaren Methoden und Eigenschaften finden Sie unter the section called “Kontext”.

Gültige Handler-Signaturen für Python-Handler

Wenn Sie Ihre Handler-Funktion in Python definieren, muss die Funktion zwei Argumente enthalten. Das erste dieser Argumente ist das Lambda-Ereignisobjekt und das zweite ist das

Lambda-Kontextobjekt. Gemäß Konvention heißen diese Eingabeargumente in der Regel `event` und `context`, Sie können ihnen jedoch beliebige Namen geben. Wenn Sie Ihre Handler-Funktion mit einem einzigen Eingabeargument deklarieren, gibt Lambda beim Versuch, Ihre Funktion auszuführen, einen Fehler aus. Die gängigste Methode zum Deklarieren einer Handler-Funktion in Python ist wie folgt:

```
def lambda_handler(event, context):
```

Sie auch können Python-Typhinweise in Ihrer Funktionsdeklaration verwenden, wie im folgenden Beispiel gezeigt:

```
from typing import Dict, Any

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, Any]:
```

Um eine bestimmte AWS Typisierung für Ereignisse zu verwenden, die von anderen generiert wurden, AWS-Services und für das Kontextobjekt, fügen Sie das `aws-lambda-typing` Paket dem Bereitstellungspaket Ihrer Funktion hinzu. Sie können diese Bibliothek in Ihrer Entwicklungsumgebung installieren, indem Sie **`pip install aws-lambda-typing`** ausführen. Der folgende Codeausschnitt zeigt, wie Sie AWS-spezifische Typhinweise verwenden. In diesem Beispiel ist das erwartete Ereignis ein Amazon-S3-Ereignis.

```
from aws_lambda_typing.events import S3Event
from aws_lambda_typing.context import Context
from typing import Dict, Any

def lambda_handler(event: S3Event, context: Context) -> Dict[str, Any]:
```

Sie können den Python-Funktionstyp `async` nicht für Ihre Handler-Funktion verwenden.

Rückgabe eines Wertes

Optional kann ein Handler einen Wert zurückgeben, der JSON-serialisierbar sein muss. Zu den gängigen Rückgabetypen gehören `dict`, `list`, `str`, `int`, `float` und `bool`.

Was mit dem zurückgegebenen Wert passiert, hängt von der Aufrufart und dem Service ab, der die Funktion aufgerufen hat. Beispiel:

- Wenn Sie den `RequestResponse` Aufruftyp verwenden, um eine Funktion synchron aufzurufen, gibt Lambda das Ergebnis des Python-Funktionsaufrufs an den Client zurück, der die Funktion

aufruft (in der HTTP-Antwort auf die Aufrufanforderung, serialisiert in JSON). Beispielsweise verwendet die AWS Lambda Konsole den `RequestResponse` Aufruftyp. Wenn Sie also die Funktion auf der Konsole aufrufen, zeigt die Konsole den zurückgegebenen Wert an.

- Wenn der Handler Objekte zurückgibt, die nicht von `json.dumps` serialisiert werden können, gibt die Laufzeit einen Fehler zurück. Eine Ausnahme ist, wenn der Handler ein `bytes` Objekt zurückgibt.
- Wenn der Handler ein `bytes` Objekt zurückgibt, leitet die Laufzeit es unverändert als rohe Binärdaten ohne JSON-Serialisierung weiter, wobei der Inhaltstyp verwendet wird. `application/unknown`
- Wenn der Handler `None` zurückgibt, wie es Python-Funktionen ohne eine `return`-Anweisung implizit tun, gibt die Laufzeit `null` zurück.
- Wenn Sie den Aufruftyp `Event` (asynchroner Aufruf) verwenden, wird der Wert verworfen.

Im Beispielcode gibt der Handler das folgende Python-Wörterbuch zurück:

```
{
  "statusCode": 200,
  "message": "Receipt processed successfully"
}
```

Die Lambda-Laufzeit serialisiert dieses Wörterbuch und gibt es als JSON-Zeichenfolge an den Client zurück, der die Funktion aufgerufen hat.

> **Note**
>
> In Python 3.9 und späteren Versionen schließt Lambda die requestId des Aufrufs in die Fehlerantwort ein.

Verwendung der AWS SDK für Python (Boto3) in Ihrem Handler

Oft verwenden Sie Ihre Funktionen, um mit anderen Ressourcen AWS-Services zu interagieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für Python (Boto3). Alle unterstützten Lambda-Python-Laufzeiten enthalten eine Version des SDK für Python (Boto3).

Wir empfehlen jedoch dringend, das SDK in das Bereitstellungspaket Ihrer Funktion aufzunehmen, wenn Ihr Code es verwenden muss. Durch das Einfügen des SDK in Ihr Bereitstellungspaket erhalten

Sie die volle Kontrolle über Ihre Abhängigkeiten und verringern das Risiko von Versionskonflikten mit anderen Bibliotheken. Weitere Informationen dazu finden Sie unter the section called “Laufzeitabhängigkeiten in Python” und the section called “Abwärtskompatibilität”.

Um das SDK for Python (Boto3) in Ihrer Funktion zu verwenden, fügen Sie dem Importblock am Anfang Ihres Funktionscodes die folgende Anweisung hinzu:

```
import boto3
```

Verwenden den `pip install`-Befehl, um Bibliothek `boto3` dem Bereitstellungspaket Ihrer Funktion hinzuzufügen. Ausführliche Anweisungen zum Hinzufügen von Abhängigkeiten zu einem ZIP-Bereitstellungspaket finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen”. Weitere Informationen zum Hinzufügen von Abhängigkeiten zu Funktionen, die als Container-Images bereitgestellt werden, finden Sie unter the section called “Erstellen eines Images aus einem Base Image” oder. the section called “Erstellen eines Images aus einem alternativen Basis-Image”

Wenn Sie `boto3` in Ihrem Code verwenden, müssen Sie keine Anmeldeinformationen angeben, um einen Client zu initialisieren. Im Beispielcode verwenden wir beispielsweise die folgende Codezeile, um einen Amazon-S3-Client zu initialisieren:

```
# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

Mit Python erstellt Lambda automatisch Umgebungsvariablen mit Anmeldeinformationen. Das `boto3` SDK überprüft die Umgebungsvariablen Ihrer Funktion bei der Initialisierung auf diese Anmeldeinformationen.

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `os.environ.get`-Methode auf Umgebungsvariablen verweisen. Im Beispielcode verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
# Access environment variables
bucket_name = os.environ.get('RECEIPT_BUCKET')
```

Vergessen Sie nicht, am Anfang Ihres Codes eine `import os`-Anweisung in den Importblock aufzunehmen.

Bewährte Codemethoden für Python-Lambda-Ereignisfunktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Funktionen die besten Programmierpraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Dies macht es einfacher, eine Funktion zu erstellen, die besser anhand von Einheiten getestet werden kann. In Python könnte das zum Beispiel so aussehen:

  ```
  def lambda_handler(event, context):
      foo = event['foo']
      bar = event['bar']
      result = my_lambda_function(foo, bar)

  def my_lambda_function(foo, bar):
      // MyLambdaFunction logic here
  ```

- Kontrollieren Sie die Abhängigkeiten im Deployment-Paket Ihrer Funktion. Die AWS Lambda - Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Node.js und Python-Laufzeiten gehören dazu die AWS SDKs.

  Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, aktualisiert Lambda diese Bibliotheken regelmäßig. Diese Updates können zu subtilen Änderungen am Verhalten Ihrer Funktion führen. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch wird die Zeit reduziert, die benötigt wird, bis Ihr Bereitstellungspaket vor dem Aufruf heruntergeladen und entpackt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit

Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates auf die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions

Der Code Ihrer AWS Lambda Funktion besteht aus einer .py-Datei, die den Handlercode Ihrer Funktion sowie alle zusätzlichen Pakete und Module enthält, von denen Ihr Code abhängt. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Dieses Paket kann entweder ein ZIP-Dateiarchiv oder ein Container-Image sein. Weitere Informationen zur Verwendung von Container-Images mit Python finden Sie unter Bereitstellen von Python-Lambda-Ereignisfunktionen mit Container-Images.

Zum Erstellen des Bereitstellungspakets für ein ZIP-Dateiarchiv können Sie ein integriertes Dienstprogramm für ZIP-Dateien Ihres Befehlszeilen-Tools oder ein anderes Dienstprogramm für ZIP-Dateien verwenden, wie 7zip. In den Beispiele in den folgenden Abschnitten wird davon ausgegangen, dass Sie ein `zip`-Befehlszeilen-Tool in einer Linux- oder MacOS-Umgebung verwenden. Um dieselben Befehle in Windows zu verwenden, können Sie das Windows-Subsystem für Linux installieren, um eine Windows-integrated Version von Ubuntu und Bash zu erhalten.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

Themen

- Laufzeitabhängigkeiten in Python
- ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen
- ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen
- Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken
- __pycache__-Ordner verwenden
- ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen
- Python-Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Laufzeitabhängigkeiten in Python

Für Lambda-Ereignisfunktionen, die die Python-Laufzeit verwenden, kann eine Abhängigkeit ein beliebiges Python-Paket oder -Modul sein. Bei Bereitstellung Ihrer Funktion mithilfe eines ZIP-Archivs können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Eine Ebene ist ein separates ZIP-Dateiarchiv, das zusätzlichen Code und andere Daten enthalten kann. Weitere Informationen zur Verwendung von Lambda-Ebenen in Python finden Sie unter the section called “Layer”.

Die Lambda-Python-Laufzeiten beinhalten die AWS SDK für Python (Boto3) und ihre Abhängigkeiten. Lambda stellt das SDK in der Laufzeit für Bereitstellungsszenarien bereit, in denen Sie keine eigenen Abhängigkeiten hinzufügen können. Zu diesen Szenarien gehören die Erstellung von Funktionen in der Konsole mithilfe des integrierten Code-Editors oder die Verwendung von Inline-Funktionen in AWS Serverless Application Model (AWS SAM) oder CloudFormation Vorlagen.

Lambda aktualisiert die Bibliotheken in der Python-Laufzeit regelmäßig, um die neuesten Updates und Sicherheitspatches aufzunehmen. Wenn Ihre Funktion die in der Runtime enthaltene Version des Boto3-SDK verwendet, Ihr Bereitstellungspaket jedoch SDK-Abhängigkeiten enthält, kann dies zu Versionsproblemen führen. Ihr Bereitstellungspaket enthält beispielsweise die SDK-Abhängigkeit urllib3. Aktualisiert Lambda das SDK während der Laufzeit, können Kompatibilitätsprobleme zwischen der neuen Version der Laufzeit und der Version von urllib3 in Ihrem Bereitstellungspaket dazu führen, dass Ihre Funktion fehlschlägt.

> **Important**
>
> Um die volle Kontrolle über Ihre Abhängigkeiten zu behalten und mögliche Probleme mit Versionsfehlern zu vermeiden, empfehlen wir Ihnen, alle Abhängigkeiten Ihrer Funktion zu Ihrem Bereitstellungspaket hinzuzufügen, auch wenn Versionen davon in der Lambda-Laufzeit enthalten sind. Dazu gehört das Boto3-SDK.

Informationen darüber, welche Version des SDK für Python (Boto3) in der von Ihnen verwendeten Laufzeit enthalten ist, finden Sie unter the section called “Runtime-included SDK-Versionen”.

Im Rahmen des AWS -Modells der geteilten Verantwortung sind Sie für die Verwaltung aller Abhängigkeiten in den Bereitstellungspaketen Ihrer Funktionen verantwortlich. Dies beinhaltet das Durchführen von Updates und Sicherheitspatches. Zum Aktualisieren von Abhängigkeiten im Bereitstellungspaket Ihrer Funktion erstellen Sie zunächst eine neue ZIP-Datei und laden Sie diese dann in Lambda hoch. Weitere Informationen finden Sie unter ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen und Python-Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen

Wenn Ihr Funktionscode keine Abhängigkeiten hat, enthält Ihre .zip-Datei nur die .py-Datei mit dem Handlercode Ihrer Funktion. Erstellen Sie mit Ihrem bevorzugten ZIP-Programm eine ZIP-Datei mit Ihrer PY-Datei im Stammverzeichnis. Wenn sich die .py-Datei nicht im Stammverzeichnis Ihrer .zip-Datei befindet, kann Lambda Ihren Code nicht ausführen.

Informationen darüber, wie Sie Ihre .zip-Datei bereitstellen, um eine neue Funktion zu erstellen oder eine bestehende zu aktualisieren, finden Sie unter. Python-Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen

Hängt Ihr Funktionscode von zusätzlichen Paketen und Modulen ab, können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Die Anweisungen in diesem Abschnitt zeigen Ihnen, wie Sie Ihre Abhängigkeiten in Ihr ZIP-Bereitstellungspaket aufnehmen. Damit Lambda Ihren Code ausführen kann, muss die PY-Datei, die Ihren Handler-Code und alle Abhängigkeiten Ihrer Funktion enthält, im Stammverzeichnis der ZIP-Datei installiert werden.

Angenommen, Ihr Funktionscode ist in einer Datei mit dem Namen „`lambda_function.py`“ gespeichert. Die folgenden CLI-Befehle erstellen Sie eine ZIP-Datei mit dem Namen „`my_deployment_package.zip`“, die Ihren Funktionscode und seine Abhängigkeiten enthält. Sie können Ihre Abhängigkeiten entweder direkt in einem Ordner in Ihrem Projektverzeichnis installieren oder eine virtuelle Python-Umgebung verwenden.

So erstellen Sie das Bereitstellungspaket (Projektverzeichnis)

1. Navigieren Sie zum Projektverzeichnis, das Ihre `lambda_function.py`-Quellcodedatei enthält. In diesem Beispiel trägt das Verzeichnis den Namen `my_function`.

   ```
   cd my_function
   ```

2. Erstellen Sie ein neues Verzeichnis mit dem Namen „package“, in das Sie Ihre Abhängigkeiten installieren.

   ```
   mkdir package
   ```

   Beachten Sie, dass Lambda bei einem ZIP-Bereitstellungspaket erwartet, dass sich Ihr Quellcode und seine Abhängigkeiten im Stammverzeichnis der ZIP-Datei befinden. Installieren Sie jedoch Abhängigkeiten direkt in Ihrem Projektverzeichnis, kann dies zu einer großen Anzahl neuer Dateien und Ordner führen und das Navigieren in Ihrer IDE erschweren. Sie erstellen hier ein separates `package`-Verzeichnis, um Ihre Abhängigkeiten von Ihrem Quellcode zu trennen.

3. Installieren Sie Ihre Abhängigkeiten im `package`-Verzeichnis. Im folgenden Beispiel wird das Boto3-SDK aus dem Python-Paketindex mithilfe von pip installiert. Verwendet Ihr Funktionscode selbst erstellte Python-Pakete, speichern Sie diese im `package`-Verzeichnis.

```
pip install --target ./package boto3
```

4. Erstellen Sie eine ZIP-Datei mit den installierten Bibliotheken im Stammverzeichnis.

```
cd package
zip -r ../my_deployment_package.zip .
```

Dadurch wird eine `my_deployment_package.zip`-Datei in Ihrem Projektverzeichnis generiert.

5. Fügen Sie die Datei „lambda_function.py“ dem Stammverzeichnis der ZIP-Datei hinzu.

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

Ihre ZIP-Datei sollte eine flache Verzeichnisstruktur haben, wobei der Handler-Code Ihrer Funktion und alle Ihre Abhängigkeitsordner wie folgt im Stammverzeichnis installiert sind.

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

Wenn sich die .py-Datei, die den Handlercode Ihrer Funktion enthält, nicht im Stammverzeichnis Ihrer .zip-Datei befindet, kann Lambda Ihren Code nicht ausführen.

So erstellen Sie das Bereitstellungspaket (virtuelle Umgebung)

1. Erstellen und aktivieren Sie eine virtuelle Umgebung in Ihrem Projektverzeichnis. In diesem Beispiel trägt das Projektverzeichnis den Namen „`my_function`“.

```
~$ cd my_function
~/my_function$ python3.14 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. Installieren Sie Ihre erforderlichen Bibliotheken mithilfe von pip. Im folgenden Beispiel wird das Boto3-SDK installiert.

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. Verwenden Sie `pip show`, um den Ort in Ihrer virtuellen Umgebung zu finden, an dem pip Ihre Abhängigkeiten installiert hat.

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

   Der Ordner, in dem Pip Ihre Bibliotheken installiert, könnte den Namen oder haben. `site-packages` `dist-packages` Dieser Ordner kann sich entweder im `lib64/python3.x` Verzeichnis `lib/python3.x` oder befinden (wobei python3.x die Version von Python darstellt, die Sie verwenden).

4. Deaktivieren Sie die virtuelle Umgebung.

```
(my_virtual_env) ~/my_function$ deactivate
```

5. Navigieren Sie in das Verzeichnis mit den Abhängigkeiten, die Sie mit pip installiert haben, und erstellen Sie eine ZIP-Datei in Ihrem Projektverzeichnis mit den installierten Abhängigkeiten im Stammverzeichnis. In diesem Beispiel hat pip Ihre Abhängigkeiten im Verzeichnis „`my_virtual_env/lib/python3.14/site-packages`“ installiert.

```
~/my_function$ cd my_virtual_env/lib/python3.14/site-packages
~/my_function/my_virtual_env/lib/python3.14/site-packages$ zip -r ../../../../
my_deployment_package.zip .
```

6. Navigieren Sie zum Stammverzeichnis Ihres Projektverzeichnisses, in dem sich die PY-Datei mit Ihrem Handler-Code befindet, und fügen Sie diese Datei dem Stammverzeichnis Ihres ZIP-Pakets hinzu. In diesem Beispiel heißt Ihre Funktionscodedatei „`lambda_function.py`“.

```
~/my_function/my_virtual_env/lib/python3.14/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken

Wenn Sie in Ihrem Code eine `import`-Anweisung verwenden, durchsucht die Python-Laufzeit die Verzeichnisse in ihrem Suchpfad, bis sie das Modul oder Paket findet. Standardmäßig ist der

erste Speicherort, den die Laufzeit durchsucht, das Verzeichnis, in das Ihr ZIP-Bereitstellungspaket entpackt und bereitgestellt wird (`/var/task`). Wenn Sie eine Version einer Bibliothek mit Runtime in Ihr Bereitstellungspaket aufnehmen, hat Ihre Version Vorrang vor der Version, die in der Runtime enthalten ist. Abhängigkeiten in Ihrem Bereitstellungspaket haben ebenfalls Vorrang vor Abhängigkeiten in Ebenen.

Wenn Sie einer Ebene eine Abhängigkeit hinzufügen, extrahiert Lambda diese in `/opt/python/lib/python3.x/site-packages` (wobei `python3.x` die von Ihnen verwendete Version der Laufzeit ist) oder in `/opt/python`. Im Suchpfad haben diese Verzeichnisse Vorrang vor den Verzeichnissen, die die in der Laufzeit enthaltenen Bibliotheken und die über pip installierten Bibliotheken enthalten (`/var/runtime` und `/var/lang/lib/python3.x/site-packages`). Bibliotheken in Funktionsebenen haben daher Vorrang vor Versionen, die in der Laufzeit enthalten sind.

> **Note**
>
> Im verwalteten Runtime- und Basis-Image von Python 3.11 sind das AWS SDK und seine Abhängigkeiten im `/var/lang/lib/python3.11/site-packages` Verzeichnis installiert.

Sie können den vollständigen Suchpfad für Ihre Funktion anzeigen, indem Sie den folgenden Codeausschnitt hinzufügen.

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> Da Abhängigkeiten in Ihrem Bereitstellungspaket oder Ihren Ebenen Vorrang vor Bibliotheken haben, die zur Laufzeit gehören, kann es zu Versionsproblemen kommen, wenn Sie eine SDK-Abhängigkeit wie urllib3 in Ihr Paket aufnehmen, ohne das SDK ebenfalls einzubetten. Stellen Sie Ihre eigene Version einer Boto3-Abhängigkeit bereit, müssen Sie Boto3 auch als Abhängigkeit in Ihrem Bereitstellungspaket bereitstellen. Wir empfehlen, dass Sie alle Abhängigkeiten Ihrer Funktion in ein Paket packen, auch wenn Versionen davon in der Laufzeit enthalten sind.

Fügen Sie Abhängigkeiten alternativ auch in einem separaten Ordner in Ihrem ZIP-Paket hinzu. Fügen Sie beispielsweise eine Version des Boto3-SDK zu einem Ordner in Ihrem ZIP-Paket mit dem Namen „`common`“ hinzu. Wird Ihr ZIP-Paket entpackt und bereitgestellt, wird dieser Ordner im Verzeichnis „`/var/task`“ abgelegt. Mit einer `import from`-Anweisung verwenden Sie eine Abhängigkeit aus einem Ordner Ihres ZIP-Bereitstellungspakets in Ihrem Code. Verwenden Sie mithilfe folgender Anweisung beispielsweise eine Version von Boto3 aus einem Ordner, der in Ihrem ZIP-Paket „`common`“ heißt.

```
from common import boto3
```

__pycache__-Ordner verwenden

Es wird empfohlen, keine `__pycache__`-Ordner in das Bereitstellungspaket Ihrer Funktion aufzunehmen. Python-Bytecode, der auf einem Build-Computer mit einer anderen Architektur oder in einem anderen Betriebssystem kompiliert wurde, ist möglicherweise nicht mit der Lambda-Ausführungsumgebung kompatibel.

ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen

Verwendet Ihre Funktion nur reine Python-Pakete und -Module, können Sie mit dem `pip install`-Befehl Ihre Abhängigkeiten auf einem beliebigen lokalen Build-Computer installieren und Ihre ZIP-Datei erstellen. Viele beliebte Python-Bibliotheken, darunter NumPy Pandas, sind kein reines Python und enthalten in C oder C++ geschriebenen Code. Wenn Sie Ihrem Bereitstellungspaket Bibliotheken hinzufügen, die C/C ++-Code enthalten, müssen Sie Ihr Paket korrekt erstellen, um sicherzustellen, dass es mit der Lambda-Ausführungsumgebung kompatibel ist.

Die meisten im Python Package Index (PyPI) verfügbaren Pakete sind als „Räder“ (.whl-Dateien) verfügbar. Eine WHL-Datei ist eine Art von ZIP-Datei, die eine erstellte Distribution mit vorkompilierten Binärdateien für ein bestimmtes Betriebssystem und eine bestimmte Befehlssatzarchitektur enthält. Um Ihr Bereitstellungspaket mit Lambda kompatibel zu machen, installieren Sie das Rad für Linux-Betriebssysteme und die Befehlssatzarchitektur Ihrer Funktion.

Einige Pakete sind möglicherweise nur als Quelldistributionen verfügbar. Für diese Pakete müssen Sie die C/C ++-Komponenten selbst kompilieren und erstellen.

Gehen Sie wie folgt vor, um anzuzeigen, welche Distributionen für Ihr erforderliches Paket verfügbar sind:

1. Suchen Sie auf der Hauptseite des Python-Paketindex nach dem Namen des Pakets.

2. Wählen Sie die Paketversion aus, die Sie verwenden möchten.
3. Wählen Sie Dateien zum Herunterladen aus.

Mit erstellten Distributionen (Wheels) arbeiten

Um ein Rad herunterzuladen, das mit Lambda kompatibel ist, verwenden Sie die Option pip`--platform`.

Wenn Ihre Funktion die x86_64-Befehlssatzarchitektur verwendet, führen Sie den folgenden `pip install` Befehl aus, um ein kompatibles Rad in Ihrem Verzeichnis zu installieren. `package` Ersetzen Sie „`--python 3.x`“ durch die Version der verwendeten Python-Laufzeit.

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Verwendet Ihre Funktion die arm64-Befehlssatzarchitektur, führen Sie den folgenden Befehl aus. Ersetzen Sie „`--python 3.x`“ durch die Version der verwendeten Python-Laufzeit.

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Mit Quelldistributionen arbeiten

Wenn Ihr Paket nur als Quelldistribution verfügbar ist, müssen Sie die C/C ++-Bibliotheken selbst erstellen. Für die Kompatibilität Ihres Pakets mit der Lambda-Ausführungsumgebung müssen Sie es in einer Umgebung erstellen, die dasselbe Amazon-Linux-Betriebssystem verwendet. Erstellen Sie dafür Ihr Paket in einer Linux-Instance von Amazon Elastic Compute Cloud (Amazon EC2).

Weitere Informationen zum Starten und Herstellen einer Verbindung mit einer Amazon-EC2-Linux-Instance finden Sie unter Erste Schritte mit Amazon EC2 im Amazon-EC2-Benutzerhandbuch.

Python-Lambda-Ereignisfunktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Sie können Ihr .zip-Paket mithilfe der Lambda-Konsole, der und der AWS Command Line Interface Lambda-API bereitstellen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Create Event Function.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.
   c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.
4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie „Event-Funktion erstellen“. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.

3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:

    a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.
    b. Klicken Sie auf Open.
    c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

ZIP-Dateifunktionen mithilfe des Konsolencode-Editors aktualisieren

Für einige Funktionen mit ZIP-Bereitstellungspaketen können Sie Ihren Funktionscode direkt mit dem in der Lambda-Konsole integrierten Code-Editor aktualisieren. Zur Verwendung dieses Features muss Ihre Funktion folgende Kriterien erfüllen:

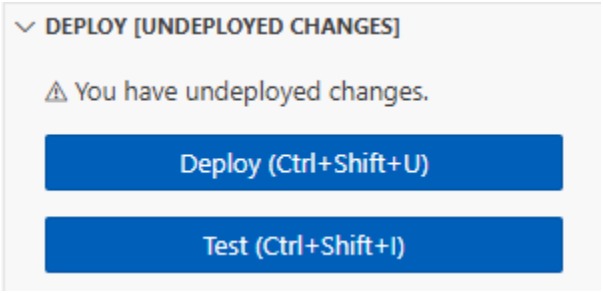
- Ihre Funktion muss eine der interpretierten Sprachlaufzeiten verwenden (Python oder Ruby) Node.js
- Das Bereitstellungspaket Ihrer Funktion muss kleiner als 50 MB (entpackt) sein.

Funktionscode für Funktionen mit Container-Image-Bereitstellungspaketen kann nicht direkt in der Konsole bearbeitet werden.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.

4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Befehle Funktion erstellen und Funktionscode aktualisieren, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre ZIP-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip-Datei mit dem zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten erstellt. Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Sie können es auch verwenden AWS SAM , um eine Lambda-Funktion mithilfe eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion zu erstellen AWS SAM, können Sie Ihre ZIP-Datei in einem Amazon S3 S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer AWS SAM Vorlage spezifiziert die `AWS::Serverless::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri`- auf die Amazon S3 S3-URI, den Pfad zum lokalen Ordner oder FunctionCodeObjekt des Funktionscodes gesetzt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Wenn Ihre ZIP-Datei größer als 50 MB ist, müssen Sie sie nicht zuerst in einen Amazon S3 S3-Bucket hochladen. AWS SAM AWS SAM kann .zip-Pakete bis zur maximal zulässigen Größe von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei in finden Sie unter AWS::Serverless: AWS SAM:Function im Entwicklerhandbuch.AWS SAM

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit CloudFormation

Sie können verwenden CloudFormation , um eine Lambda-Funktion mithilfe eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei

zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

Für Node.js und Python-Laufzeiten können Sie auch Inline-Quellcode in Ihrer CloudFormation Vorlage bereitstellen. CloudFormation erstellt dann eine ZIP-Datei, die Ihren Code enthält, wenn Sie Ihre Funktion erstellen.

Eine vorhandene ZIP-Datei verwenden

In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Eine ZIP-Datei aus dem Inline-Code erstellen

Sie können einfache Funktionen, die in Python oder Node.js Inline geschrieben sind, in einer CloudFormation Vorlage deklarieren. Da der Code in YAML oder JSON eingebettet ist, können Sie Ihrem Bereitstellungspaket keine externen Abhängigkeiten hinzufügen. Das bedeutet, dass Ihre Funktion die Version des AWS SDK verwenden muss, die in der Laufzeit enthalten ist. Die Anforderungen der Vorlage, wie die Notwendigkeit von Escapezeichen für bestimmte Zeichen, erschweren ebenfalls die Verwendung von IDE-Features zur Syntaxprüfung und Codevervollständigung. Ihre Vorlage erfordert deshalb möglicherweise zusätzliche Tests. Aufgrund dieser Einschränkungen eignet sich die Inline-Deklaration von Funktionen am besten für sehr einfachen Code, der sich nicht häufig ändert.

Um eine .zip-Datei aus Inline-Code für Node.js und Python-Laufzeiten zu erstellen, legen Sie die folgenden Eigenschaften in der Ressource Ihrer Vorlage fest: `AWS::Lambda::Function`

- `PackageType` – festlegen auf `Zip`
- `Code` – Funktionscode in das `ZipFile`-Feld eingeben
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die CloudFormation generierte ZIP-Datei darf 4 MB nicht überschreiten. Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei finden Sie unter AWS::Lambda: :Function im Benutzerhandbuch. CloudFormationCloudFormation

Stellen Sie Python-Lambda-Ereignisfunktionen mit Container-Images bereit

Es gibt drei Möglichkeiten, ein Container-Image für eine Python-Lambda-Ereignisfunktion zu erstellen:

- Mit einem AWS Basisbild für Python

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Mit einem AWS OS-only Basisbild

  AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Python in das Image aufnehmen.

- Verwenden Sie ein AWS Nicht-Basis-Image

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Python in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basisbilder für Python

- Mit einem AWS Basisbild für Python
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basisbilder für Python

AWS stellt die folgenden Basis-Images für Python bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
3.15	Python 3.15	Amazon Linux 2023	Dockerfile für Python 3.15 auf GitHub	Nicht geplant
3.14	Python 3.14	Amazon Linux 2023	Dockerfile für Python 3.14 auf GitHub	30. Juni 2029
3.13	Python 3.13	Amazon Linux 2023	Dockerfile für Python 3.13 auf GitHub	30. Juni 2029
3,12	Python 3.12	Amazon Linux 2023	Dockerfile für Python 3.12 auf GitHub	31. Oktober 2028
3.11	Python 3.11	Amazon Linux 2	Dockerfile für Python 3.11 auf GitHub	30. Juni 2027
3,10	Python 3.10	Amazon Linux 2	Dockerfile für Python 3.10 auf GitHub	31. Oktober 2026

Amazon ECR-Repository: gallery.ecr. aws/lambda/python

Die Basis-Images für Python 3.12 und höher basieren auf dem Amazon Linux 2023 Minimal Container Image. Die Python 3.8-3.11-Basis-Images basieren auf dem Amazon Linux 2-Image. AL2023-based Images bieten mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie`glibc`.

AL2023-based Bilder verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, was der Standard-Paketmanager in Amazon Linux 2 ist. `microdnf`ist eine eigenständige

Implementierung von. `dnf` Eine Liste der Pakete, die in AL2023-based Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2023 und Amazon Linux 2 finden Sie unter Introducing the Amazon Linux 2023 runtime for AWS Lambda im AWS -Compute-Blog.

> **Note**
>
> Um AL2023-based Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Suchpfad für Abhängigkeiten in den Basis-Images

Wenn Sie in Ihrem Code eine `import`-Anweisung verwenden, durchsucht die Python-Laufzeit die Verzeichnisse in ihrem Suchpfad, bis sie das Modul oder Paket findet. Standardmäßig durchsucht die Laufzeit zuerst das Verzeichnis `{LAMBDA_TASK_ROOT}`. Wenn Sie eine Version einer in der Laufzeit enthaltenen Bibliothek in Ihr Image einfügen, hat Ihre Version Vorrang vor der Version, die in der Laufzeit enthalten ist.

Andere Schritte im Suchpfad hängen davon ab, welche Version des Lambda-Basis-Images für Python Sie verwenden:

- Python 3.11 und höher: Runtime-included Bibliotheken und über Pip installierte Bibliotheken werden im Verzeichnis installiert. `/var/lang/lib/python3.11/site-packages` Dieses Verzeichnis hat im Suchpfad Vorrang vor `/var/runtime`. Sie können das SDK außer Kraft setzen, indem Sie pip verwenden, um eine neuere Version zu installieren. Sie können mithilfe von pip überprüfen, ob das in der Laufzeit enthaltene SDK und seine Abhängigkeiten mit allen Paketen kompatibel sind, die Sie installieren.
- Python 3.8-3.10: Runtime-included Bibliotheken sind im Verzeichnis installiert. `/var/runtime` Pip-installed Bibliotheken sind im Verzeichnis installiert. `/var/lang/lib/python3.x/site-packages` Das Verzeichnis `/var/runtime` hat im Suchpfad Vorrang vor `/var/lang/lib/python3.x/site-packages`.

Sie können den vollständigen Suchpfad für Ihre Funktion einsehen, indem Sie den folgenden Codeausschnitt hinzufügen.

```
import sys
```

```
search_path = sys.path
print(search_path)
```

Mit einem AWS Basisbild für Python

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Python

Erstellen eines Images aus einem Base Image

Um ein Container-Image aus einem zu erstellen AWS Basisbild für Python

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.py`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Python-Funktion

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `requirements.txt`. Wenn Sie den Beispielfunktionscode aus dem vorherigen Schritt verwenden, können Sie die Datei leer lassen, da es keine Abhängigkeiten gibt. Andernfalls listen Sie jede benötigte Bibliothek auf. So sollte beispielsweise Ihre `requirements.txt` aussehen, wenn Ihre Funktion AWS SDK für Python (Boto3) verwendet:

Example requirements.txt

```
boto3
```

4. Erstellen Sie eine neue Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die FROM-Eigenschaft auf den URI des Basis-Images.
   - Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in eine Lambda-defined Umgebungsvariable zu {LAMBDA_TASK_ROOT} kopieren.
   - Legen Sie das CMD-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/python:3.12

# Copy requirements.txt
COPY requirements.txt ${LAMBDA_TASK_ROOT}

# Install the specified packages
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

- Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
- Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

   > **Note**
   >
   > Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-only Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Runtime-Interface-Client in Ihr Image aufnehmen. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Laufzeitschnittstellen-Client für Python mit dem pip-Paketmanager:

```
pip install awslambdaric
```

Sie können den Python-Runtime-Interface-Client auch von herunterladen GitHub.

Das folgende Beispiel zeigt, wie ein Container-Image für Python mit einem AWS Nicht-Base-Image erstellt wird. Das Beispiel-Dockerfile verwendet ein offizielles Python-Basis-Image. Das Dockerfile enthält den Laufzeitschnittstellen-Client für Python.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Python

Erstellen eines Images aus einem alternativen Basis-Image

Um ein Container-Image aus einem Nicht-AWS Basis-Image

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.py`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Python-Funktion

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `requirements.txt`. Wenn Sie den Beispielfunktionscode aus dem vorherigen Schritt verwenden, können Sie die Datei leer lassen, da es keine Abhängigkeiten gibt. Andernfalls listen Sie jede benötigte Bibliothek auf. So sollte beispielsweise Ihre `requirements.txt` aussehen, wenn Ihre Funktion AWS SDK für Python (Boto3) verwendet:

Example requirements.txt

```
boto3
```

4. Erstellen Sie eine neue Docker-Datei. Das folgende Dockerfile verwendet ein offizielles Python-Basis-Image anstelle eines AWS -Basis-Images. Das Dockerfile enthält den Laufzeitschnittstellen-Client, der das Image mit Lambda kompatibel macht. Die folgende Beispiel-Docker-Datei verwendet eine mehrstufige Entwicklung.

   - Legen Sie die `FROM`-Eigenschaft auf das Basis-Image fest.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie `CMD` auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM python:3.12 AS build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric
```

```
# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.

- `/usr/local/bin/python -m awslambdaric lambda_function.handler` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/python -m awslambdaric lambda_function.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/python -m awslambdaric lambda_function.handler
```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

Linux/macOS

Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

- Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
- Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

   > **Note**
   >
   > Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Ein Beispiel für das Erstellen eines Python-Images aus einem Alpine-Basis-Image finden Sie unter Unterstützung für Container-Images für Lambda im AWS Blog.

Arbeiten mit Ebenen für Python-Lambda-Ereignisfunktionen

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Fügen Sie die Ebene zu Ihrer Funktion hinzu
- Beispiel-App

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Erstellen Sie den Layer mit derselben Python-Version, die Sie für die Funktion verwenden möchten. Wenn Sie Ihren Layer beispielsweise mit Python 3.14 erstellen, verwenden Sie die Python 3.14-Laufzeit für Ihre Funktion.
- Ihre ZIP-Datei muss ein `python`-Verzeichnis auf der Stammebene enthalten.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder mit `pip` installierte Python-Bibliotheken von Drittanbietern (wie `requests` oder `pandas`) oder Ihre eigenen Python-Module und -Pakete enthalten.

Third-party Abhängigkeiten

So erstellen Sie eine Ebene mit pip-Paketen

1. Wählen Sie eine der folgenden Methoden, um `pip`-Pakete in das erforderliche Verzeichnis der obersten Ebene (`python/`) zu installieren:

   pip install

   Für reine Python-Pakete (wie Anfragen oder boto3):

   ```
   pip install requests -t python/
   ```

   Einige Python-Pakete, wie NumPy Pandas, enthalten kompilierte C-Komponenten. Wenn Sie eine Ebene mit diesen Paketen auf macOS oder Windows erstellen, müssen Sie möglicherweise diesen Befehl verwenden, um ein Linux-compatible Rad zu installieren:

   ```
   pip install numpy --platform manylinux2014_x86_64 --only-binary=:all: -t python/
   ```

   Weitere Informationen zur Arbeit mit Python-Paketen, die kompilierte Komponenten enthalten, finden Sie unter ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen.

   requirements.txt

   Mithilfe einer `requirements.txt`-Datei können Sie Paketversionen verwalten und konsistente Installationen sicherstellen.

   Example requirements.txt

   ```
   requests==2.31.0
   boto3==1.37.34
   numpy==1.26.4
   ```

   Wenn Ihre `requirements.txt`-Datei nur reine Python-Pakete enthält (wie Anfragen oder boto3):

   ```
   pip install -r requirements.txt -t python/
   ```

Einige Python-Pakete, wie NumPy Pandas, enthalten kompilierte C-Komponenten. Wenn Sie eine Ebene mit diesen Paketen auf macOS oder Windows erstellen, müssen Sie möglicherweise diesen Befehl verwenden, um ein Linux-compatible Rad zu installieren:

```
pip install -r requirements.txt --platform manylinux2014_x86_64 --only-
binary=:all: -t python/
```

Weitere Informationen zur Arbeit mit Python-Paketen, die kompilierte Komponenten enthalten, finden Sie unter ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen.

2. Zippen Sie den Inhalt des `python`-Verzeichnisses.

Linux/macOS

```
zip -r layer.zip python/
```

PowerShell

```
Compress-Archive -Path .\python -DestinationPath .\layer.zip
```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
python/               # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

> **Note**
>
> Wenn Sie eine virtuelle Python-Umgebung (venv) verwenden, um Pakete zu installieren, ist Ihre Verzeichnisstruktur anders (z. B.`python/lib/python3.x/site-packages`). Solange Ihre ZIP-Datei das `python`-Verzeichnis auf der Stammebene enthält, kann Lambda Ihre Pakete finden und importieren.

Benutzerdefinierte Python-Module

So erstellen Sie eine Ebene mithilfe Ihres eigenen Codes

1. Erstellen Sie das erforderliche Verzeichnis der obersten Ebene für Ihre Ebene:

```
mkdir python
```

2. Erstellen Sie Ihre Python-Module im `python`-Verzeichnis. Das folgende Beispielmodul validiert Bestellungen, indem es überprüft, ob sie die erforderlichen Informationen enthalten.

   Example Benutzerdefiniertes Modul: validator.py

```
import json

def validate_order(order_data):
    """Validates an order and returns formatted data."""
    required_fields = ['product_id', 'quantity']

    # Check required fields
    missing_fields = [field for field in required_fields if field not in
 order_data]
    if missing_fields:
        raise ValueError(f"Missing required fields: {', '.join(missing_fields)}")

    # Validate quantity
    quantity = order_data['quantity']
    if not isinstance(quantity, int) or quantity < 1:
        raise ValueError("Quantity must be a positive integer")

    # Format and return the validated data
    return {
        'product_id': str(order_data['product_id']),
        'quantity': quantity,
        'shipping_priority': order_data.get('priority', 'standard')
    }

def format_response(status_code, body):
    """Formats the API response."""
    return {
        'statusCode': status_code,
        'body': json.dumps(body)
```

```
    }
```

3. Zippen Sie den Inhalt des `python`-Verzeichnisses.

   Linux/macOS

   ```
   zip -r layer.zip python/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\python -DestinationPath .\layer.zip
   ```

   Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

   ```
   python/     # Required top-level directory
   ### validator.py
   ```

4. Importieren und verwenden Sie die Module in Ihrer Funktion wie bei jedem anderen Python-Paket. Beispiel:

   ```
   from validator import validate_order, format_response
   import json

   def lambda_handler(event, context):
       try:
           # Parse the order data from the event body
           order_data = json.loads(event.get('body', '{}'))

           # Validate and format the order
           validated_order = validate_order(order_data)

           return format_response(200, {
               'message': 'Order validated successfully',
               'order': validated_order
           })
       except ValueError as e:
           return format_response(400, {
               'error': str(e)
           })
       except Exception as e:
           return format_response(500, {
   ```

```
            'error': 'Internal server error'
        })
```

Sie können das folgende Testereignis verwenden, um die Funktion aufzurufen:

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Erwartete Antwort:

```
{
  "statusCode": 200,
  "body": "{\"message\": \"Order validated successfully\", \"order\": {\"product_id
\": \"ABC123\", \"quantity\": 2, \"shipping_priority\": \"express\"}}"
}
```

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den AWS CLI Befehl publish-layer-version aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes python3.14
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.

4. (Optional) Wählen Sie für Kompatible Laufzeiten die Python-Laufzeit aus, die der Python-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Um die Ebene an Ihre Funktion anzuhängen, führen Sie den Befehl update-function-configuration aus. AWS CLI Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

> **Note**
>
> Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Beispiel-App

Weitere Beispiele für die Verwendung von Lambda-Layern finden Sie in der Layer-Python-Beispielanwendung im AWS Lambda Developer Guide-Repository. GitHub Diese Anwendung umfasst zwei Ebenen, die Python-Bibliotheken enthalten. Nach dem Erstellen der Ebenen können Sie die entsprechende Funktionen bereitstellen und aufrufen, um zu bestätigen, dass die Ebenen wie erwartet funktionieren.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Python-Ereignisfunktion

Wenn Lambda Ihre Event-Funktion ausführt, übergibt es ein Kontextobjekt an den Handler. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit. Weitere Informationen darüber, wie das Kontextobjekt an den Funktions-Handler übergeben wird, finden Sie unter Definieren Sie den Lambda-Ereignisfunktionshandler in Python.

Context-Methoden

- `get_remaining_time_in_millis` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `function_name` – Der Name der Lambda-Funktion.
- `function_version` – Die Version der Funktion.
- `invoked_function_arn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memory_limit_in_mb` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `aws_request_id` – Der Bezeichner der Aufrufanforderung.
- `log_group_name` – Protokollgruppe für die Funktion.
- `log_stream_name` – Der Protokollstream für die Funktions-Instance.
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
  - `cognito_identity_id`— Die authentifizierte Amazon-Cognito-Identität.
  - `cognito_identity_pool_id` – Der Amazon-Cognito-Identitätspool, der den Aufruf autorisiert hat.
- `client_context` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`

- `client.app_package_name`
- `custom` – Ein `dict` mit benutzerdefinierten Werten, die von der mobilen Clientanwendung festgelegt wurden.
- `env`— Eine Reihe `dict` von Umgebungsinformationen, die vom AWS SDK bereitgestellt werden.

Powertools für Lambda (Python) bietet eine Schnittstellendefinition für das Lambda-Kontextobjekt. Sie können die Schnittstellendefinition für Typhinweise verwenden oder die Struktur des Lambda-Kontextobjekts genauer untersuchen. Die Schnittstellendefinition finden Sie unter lambda_context.py im Powertools-Lambda-Python-Repository unter. GitHub

Das folgende Beispiel zeigt eine Handler-Funktion zur Protokollierung von Context-Informationen.

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

Zusätzlich zu den oben aufgeführten Optionen können Sie das AWS X-Ray SDK auch verwenden, um kritische Codepfade Instrumentierung von Python-Code in AWS Lambda zu identifizieren, ihre Leistung zu verfolgen und die Daten für die Analyse zu erfassen.

Python-Lambda-Ereignisfunktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterLambda Event Function-Logs an CloudWatch Logs senden.

Zur Ausgabe von Protokollen aus Ihrem Funktionscode können Sie das integrierte `logging`-Modul verwenden. Für detailliertere Einträge können Sie jede Protokollierungsbibliothek verwenden, die in `stdout` oder `stderr` schreibt.

Ausdrucken in das Protokoll

Um eine grundlegende Ausgabe an die Protokolle zu senden, können Sie in Ihrer Funktion eine `print`-Methode verwenden. Im folgenden Beispiel werden die Werte der CloudWatch Loggruppe und des Streams Logs sowie das Event-Objekt protokolliert.

Beachten Sie, dass Lambda Protokollausgaben nur im Klartextformat an Logs senden kann, wenn Ihre Funktion CloudWatch Logs mithilfe von `print` Python-Anweisungen ausgibt. Um Protokolle in strukturiertem JSON zu erfassen, müssen Sie eine unterstützte Protokollierungsbibliothek verwenden. Weitere Informationen finden Sie unter the section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Python”.

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example Protokollausgabe

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2025/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
```

```
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Die Python-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die `REPORT`-Zeile enthält die folgenden Daten:

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Verwendung einer Protokollierungsbibliothek

Für detailliertere Protokolle verwenden Sie das Modul zur Protokollierung in der Standardbibliothek oder eine beliebige Protokollierungsbibliothek eines Drittanbieters, die in `stdout` oder `stderr` schreibt.

Für unterstützte Python-Laufzeiten können Sie wählen, ob mit dem `logging`-Standardmodul erstellte Protokolle im Klartext oder in JSON erfasst werden. Weitere Informationen hierzu finden Sie unter the section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Python”.

Derzeit ist das Standard-Protokollformat für alle Python-Laufzeiten das Klartextformat. Das folgende Beispiel zeigt, wie Protokollausgaben, die mit dem `logging` Standardmodul erstellt wurden, im Klartext in CloudWatch Logs erfasst werden.

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

Die Ausgabe aus `logger` umfasst die Protokollebene, den Zeitstempel und die Anforderungs-ID.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2025/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

> **Note**
>
> Wenn das Protokollformat Ihrer Funktion auf Klartext gesetzt ist, lautet die Standardeinstellung auf Protokollebene für Python-Laufzeiten WARN. Das bedeutet, dass Lambda nur Protokollausgaben der Stufe WARN und niedriger an CloudWatch Logs sendet. Um die Standardprotokollebene zu ändern, verwenden Sie die Python-

Methode `logging setLevel()`, wie in diesem Beispielcode gezeigt. Wenn Sie das Protokollformat Ihrer Funktion auf JSON setzen, empfehlen wir, die Protokollebene der Funktion mithilfe der erweiterten Lambda Advanced Logging Controls zu konfigurieren und nicht die Protokollebene im Code festzulegen. Weitere Informationen hierzu finden Sie unter the section called “Verwenden der Filterung auf Protokollebene mit Python”.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Python

Um Ihnen mehr Kontrolle darüber zu geben, wie die Logs Ihrer Funktionen erfasst, verarbeitet und genutzt werden, können Sie die folgenden Protokollierungsoptionen für unterstützte Lambda-Python-Laufzeiten konfigurieren:

- Protokollformat — wählen Sie zwischen Klartext und strukturiertem JSON-Format für die Logs Ihrer Funktion
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, die Lambda an Amazon sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Protokolle sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.

Weitere Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Python-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden strukturierter JSON-Protokolle mit Python

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die von der Python-Standardprotokollierungsbibliothek ausgegebenen Protokolle CloudWatch als strukturiertes JSON an. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Die `logging`-Bibliothek von Python kann auch zusätzliche Schlüssel-Wert-Paare wie `"logger"` zu diesem JSON-Objekt hinzufügen.

Die Beispiele in den folgenden Abschnitten zeigen, wie mit der `logging` Python-Bibliothek generierte Protokollausgaben in CloudWatch Logs erfasst werden, wenn Sie das Protokollformat Ihrer Funktion als JSON konfigurieren.

Beachten Sie, dass Lambda, wenn Sie die `print` Methode verwenden, um grundlegende Protokollausgaben zu erzeugenthe section called “Ausdrucken in das Protokoll”, wie unter beschrieben, diese Ausgaben als Klartext erfasst, auch wenn Sie das Protokollierungsformat Ihrer Funktion als JSON konfigurieren.

Standard-JSON-Protokollausgaben mithilfe der Python-Protokollierungsbibliothek

Der folgende Beispielcodeausschnitt und die Protokollausgabe zeigen, wie mit der `logging` Python-Bibliothek generierte Standardprotokollausgaben in CloudWatch Logs erfasst werden, wenn das Protokollformat Ihrer Funktion auf JSON gesetzt ist.

Example Python-Protokollierungscode

```
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON-Protokolldatensatz

```
{
    "timestamp":"2025-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Protokollieren zusätzlicher Parameter in JSON

Wenn das Protokollformat Ihrer Funktion auf JSON eingestellt ist, können Sie auch zusätzliche Parameter mit der standardmäßigen Python `logging`-Bibliothek protokollieren, indem Sie das Schlüsselwort `extra` verwenden, um ein Python-Wörterbuch an die Protokollausgabe zu übergeben.

Example Python-Protokollierungscode

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON-Protokolldatensatz

```
{
  "timestamp": "2025-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
}
```

Protokollierungsausnahmen in JSON

Der folgende Codeausschnitt zeigt, wie Python-Ausnahmen in der Protokollausgabe Ihrer Funktion erfasst werden, wenn Sie das Protokollformat als JSON konfigurieren. Beachten Sie, dass den mit `logging.exception` generierten Protokollausgaben die Protokollebene ERROR zugewiesen wird.

Example Python-Protokollierungscode

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON-Protokolldatensatz

```
{
  "timestamp": "2025-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

Strukturierte JSON-Protokolle mit anderen Protokollierungstools

Wenn Ihr Code bereits eine andere Logging-Bibliothek wie Powertools for verwendet AWS Lambda, um strukturierte JSON-Logs zu erstellen, müssen Sie keine Änderungen vornehmen. AWS Lambda codiert keine Logs doppelt, die bereits JSON-kodiert sind. Selbst wenn Sie Ihre Funktion so konfigurieren, dass sie das JSON-Protokollformat verwendet, erscheinen Ihre Logging-Ausgaben CloudWatch in der von Ihnen definierten JSON-Struktur.

Das folgende Beispiel zeigt, wie Protokollausgaben, die mit dem AWS Lambda Paket Powertools for generiert wurden, in CloudWatch Logs erfasst werden. Das Format dieser Protokollausgabe ist dasselbe, unabhängig davon, ob die Protokollierungskonfiguration Ihrer Funktion auf JSON oder TEXT gesetzt ist. Weitere Informationen zur Verwendung von Powertools für finden Sie AWS Lambda unter the section called “Verwendung von Powertools für AWS Lambda (Python) und AWS SAM für strukturiertes Logging” und the section called “Verwenden von Powertools für AWS Lambda (Python) und AWS CDK für strukturiertes Logging”

Example Codeausschnitt für die Python-Protokollierung (mit Powertools für AWS Lambda)

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON-Protokolldatensatz (mit Powertools für AWS Lambda)

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2025-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

Verwenden der Filterung auf Protokollebene mit Python

Durch die Konfiguration der Filterung auf Protokollebene können Sie festlegen, dass nur Protokolle mit einer bestimmten Protokollierungsebene oder niedriger an Logs gesendet werden. CloudWatch Informationen zur Konfiguration der Filterung auf Protokollebene für Ihre Funktion finden Sie unter the section called “Log-level filtern”.

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mit der `logging` Standard-Python-Bibliothek und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. AWS Lambda filtert dann Ihre Protokollausgaben mithilfe des Schlüsselwertpaars „Level“ im JSON-Objekt, wie unter beschriebenthe section called “Verwenden strukturierter JSON-Protokolle mit Python”. Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unterthe section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.
- Verwenden Sie eine andere Logging-Bibliothek oder Methode, um strukturierte JSON-Logs in Ihrem Code zu erstellen, die ein „Level“ -Schlüsselwertpaar enthalten, das die Ebene der Protokollausgabe definiert. Sie können Powertools beispielsweise verwenden, AWS Lambda um strukturierte JSON-Protokollausgaben aus Ihrem Code zu generieren.

  Sie können auch eine „print“-Anweisung verwenden, um ein JSON-Objekt auszugeben, das eine Protokollebenenkennung enthält. Die folgende Print-Anweisung erzeugt eine Ausgabe im JSON-Format, bei der die Protokollebene auf INFO gesetzt ist. AWS Lambda sendet das JSON-Objekt an CloudWatch Logs, wenn die Protokollierungsebene Ihrer Funktion auf INFO, DEBUG oder TRACE gesetzt ist.

  ```
  print('{"msg":"My log message", "level":"info"}')
  ```

Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in CloudWatch der Konsole anzeigen

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Logs für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Protokolle anzeigen mit AWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Andere Tools und Bibliotheken für die Protokollierung verwenden

Powertools for AWS Lambda (Python) ist ein Entwickler-Toolkit zur Implementierung serverloser Best Practices und zur Steigerung der Entwicklergeschwindigkeit. Das Logger-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen über den Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)
- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu
- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (Python) und AWS SAM für strukturiertes Logging

Führen Sie die nachstehenden Schritte aus, um eine Hello-World-Python-Beispielanwendung mit integrierten -Powertools for Python-Modulen unter Verwendung von AWS SAM bereitzustellen. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.9
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mit der Hello World Python-Vorlage.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

Das Ergebnis sieht folgendermaßen aus:

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
```

```
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
```

```
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Verwenden von Powertools für AWS Lambda (Python) und AWS CDK für strukturiertes Logging

Gehen Sie wie folgt vor, um eine Hello World Python-Beispielanwendung mit integrierten Powertools for AWS Lambda (Python) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den AWS CDK. Diese Anwendung implementiert ein grundlegendes API-Backend und

verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.9
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS CDK Anwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language python
```

3. Installieren Sie die Python-Abhängigkeiten.

```
pip install -r requirements.txt
```

4. Erstellen Sie ein Verzeichnis lambda_function unter dem Stammordner.

```
mkdir lambda_function
cd lambda_function
```

5. Erstellen Sie eine Datei namens app.py und fügen Sie den folgenden Code zur Datei hinzu. Dies ist der Code für die -Funktion.

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
```

```
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. Öffnen Sie das Verzeichnis hello_world. Sie sollten eine Datei mit dem Namen hello_world_stack.py sehen.

```
cd ..
cd hello_world
```

7. Öffnen Sie hello_world_stack.py und fügen Sie den folgenden Code in die Datei ein. Dies enthält den Lambda-Konstruktor, der die Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt, und den ApiGatewayv1-Konstruktor, der die REST-API erstellt.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # Using AWS Lambda Powertools with Lambda Layer
            # This imports the Powertools layer which provides observability
 features for Lambda functions
            # For available versions, see: https://docs.aws.amazon.com/powertools/
python/latest/#lambda-layer
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
```

```
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Stellen Sie die Anwendung bereit.

```
cd ..
cdk deploy
```

9. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Rufen Sie den API-Endpunkt auf:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

    Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

11. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name HelloWorldStack
```

    Das Ergebnis sieht folgendermaßen aus:

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
```

```
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
```

```
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

AWS Lambda Testen von Event-Funktionen in Python

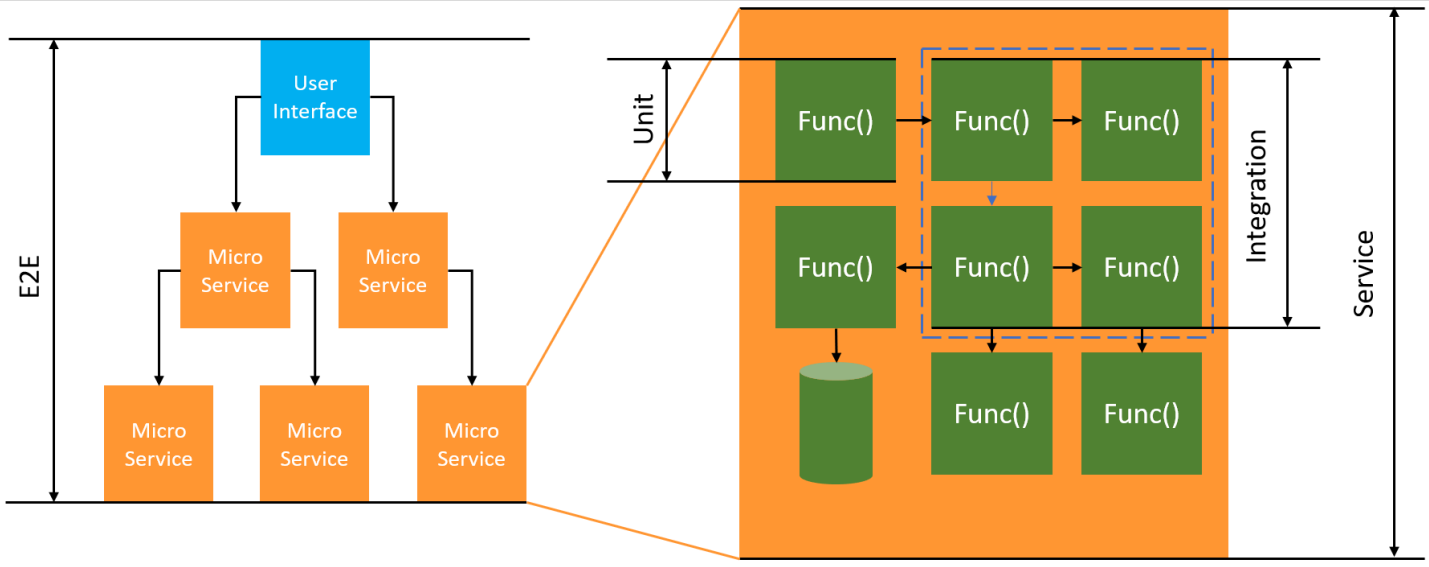
> **Note**
>
> Im Kapitel Testen von Funktionen finden Sie eine vollständige Einführung in Techniken und bewährte Methoden für das Testen von Serverless-Lösungen.

Beim Testen serverloser Funktionen werden herkömmliche Testtypen und -techniken verwendet. Sie müssen jedoch auch erwägen, serverlose Anwendungen als Ganzes zu testen. Cloud-based Tests bieten das genaueste Maß für die Qualität Ihrer Funktionen und serverlosen Anwendungen.

Eine Serverless-Anwendungsarchitektur umfasst verwaltete Services, die über API-Aufrufe wichtige Anwendungsfunktionen bereitstellen. Aus diesem Grund muss Ihr Entwicklungszyklus automatisierte Tests beinhalten, die bei der Interaktion Ihrer Funktionen und Services die Funktionalität überprüfen.

Wenn Sie keine cloud-basierten Tests erstellen, können aufgrund von Unterschieden zwischen Ihrer lokalen Umgebung und der bereitgestellten Umgebung Probleme auftreten. Ihr kontinuierlicher Integrationsprozess muss Tests anhand einer Reihe von Ressourcen durchführen, die in der Cloud bereitgestellt werden, bevor Ihr Code in die nächste Bereitstellungsumgebung wie QA, Staging oder Produktion übertragen wird.

Lesen Sie diesen kurzen Leitfaden weiter, um weitere Informationen zu Teststrategien für Serverless-Anwendungen zu erhalten, oder besuchen Sie das Serverless Test Samples Repository, um praktische Beispiele zu finden, die sich speziell auf die gewählte Sprache und Laufzeit beziehen.



Für serverlose Tests schreiben Sie immer noch Einheiten -, Integrations - und End-to-End-Tests.

- Einheitentests — Tests, die an einem isolierten Codeblock ausgeführt werden. Zum Beispiel die Überprüfung der Geschäftslogik zur Berechnung der Bereitstellungskosten für einen bestimmten Artikel und Bestimmungsort.
- Integrationstests — Tests, an denen zwei oder mehr Komponenten oder Dienste beteiligt sind, die interagieren, in der Regel in einer Cloud-Umgebung. Bei der Überprüfung einer Funktion werden beispielsweise Ereignisse aus einer Warteschlange verarbeitet.
- End-to-end Tests — Tests, die das Verhalten einer gesamten Anwendung überprüfen. Stellen Sie beispielsweise sicher, dass die Infrastruktur korrekt eingerichtet ist und die Ereignisse zwischen den Services wie erwartet ablaufen, um die Bestellungen der Kunden aufzuzeichnen.

Testen Ihrer Serverless-Anwendungen

Im Allgemeinen verwenden Sie verschiedene Methoden, um Ihren serverlosen Anwendungscode zu testen, darunter Tests in der Cloud, Tests mit Mocks und gelegentlich Tests mit Emulatoren.

Testen in der Cloud

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Sie führen Tests für Code durch, der in der Cloud bereitgestellt wird und mit cloud-basierten Services interagiert. Dieser Ansatz bietet das genaueste Maß für die Qualität Ihres Codes.

Eine bequeme Möglichkeit, Ihre Lambda Event Function in der Cloud zu debuggen, ist die Konsole mit einem Testereignis. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres JSON-Dokument (`{}`) sein. Die Konsole bietet Beispielereignisse für eine Vielzahl von Service-Integrationen. Nachdem Sie ein Ereignis in der Konsole erstellt haben, können Sie es mit Ihrem Team teilen, um das Testen einfacher und einheitlicher zu gestalten.

> **Note**
>
> Das Testen einer Funktion in der Konsole ist ein schneller Einstieg, aber die Automatisierung Ihrer Testzyklen gewährleistet die Anwendungsqualität und die Entwicklungsgeschwindigkeit.

Test-Tools

Es gibt Tools und Techniken, um die Feedback-Schleifen bei der Entwicklung zu beschleunigen. AWS SAM Accelerate und AWS CDK Watch Mode reduzieren beispielsweise beide die Zeit, die für die Aktualisierung von Cloud-Umgebungen benötigt wird.

Moto ist eine Python-Bibliothek zum Nachahmen von AWS Diensten und Ressourcen, sodass Sie Ihre Funktionen mit wenigen oder keinen Änderungen testen können, indem Sie Dekoratoren verwenden, um Antworten abzufangen und zu simulieren.

Die Validierungsfunktion der Powertools for AWS Lambda (Python) bietet Dekoratoren, mit denen Sie Eingabeereignisse und Ausgabeantworten Ihrer Python-Funktionen validieren können.

Weitere Informationen finden Sie im Blogbeitrag Unit Testing Lambda with Python and Mock AWS Services.

Informationen zur Reduzierung der Latenz bei Iterationen der Cloud-Bereitstellung finden Sie unter AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) Watch-Modus. Diese Tools überwachen Ihre Infrastruktur und Ihren Code auf Änderungen. Sie reagieren auf diese Änderungen, indem sie automatisch inkrementelle Updates erstellen und in Ihrer Cloud-Umgebung bereitstellen.

Beispiele, in denen diese Tools verwendet werden, sind im Code-Repository mit Python-Testbeispielen verfügbar.

Instrumentierung von Python-Code in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können es verwenden X-Ray, um eine Anfrage zu verfolgen, während sie Ressourcen in Ihrer Anwendung durchquert, zu denen Lambda-Funktionen und andere AWS Dienste gehören können.

Um Ablaufverfolgungsdaten an zu senden X-Ray, können Sie eine von drei SDK-Bibliotheken verwenden:

- AWS Distribution for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des (OTel) SDK. OpenTelemetry
- AWS X-Ray-SDK für Python— Ein SDK zum Generieren und Senden von Trace-Daten an. X-Ray
- Powertools for AWS Lambda (Python) — Ein Entwickler-Toolkit zur Implementierung serverloser Best Practices und zur Steigerung der Entwicklergeschwindigkeit.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den Dienst zu senden. X-Ray Anschließend können Sie X-Ray die Leistungskennzahlen Ihrer Anwendung anzeigen, filtern und Einblicke in sie gewinnen, um Probleme und Optimierungsmöglichkeiten zu identifizieren.

> **Important**
>
> Die SDKs X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von angeboten wird. AWS Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgängige Ablaufverfolgung implementieren. X-Ray Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Auswahl zwischen der AWS Distribution for Open Telemetry und SDKs. X-Ray

Sections

- Verwenden von Powertools für AWS Lambda (Python) und AWS SAM für die Rückverfolgung
- Verwenden Sie Powertools für AWS Lambda (Python) und die AWS CDK für die Rückverfolgung
- Verwenden von ADOT zum Instrumentieren Ihrer Python-Funktionen
- Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Python-Funktionen

- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Die Ablaufverfolgung wird aktiviert mit CloudFormation
- Interpretieren und X-Ray Trace
- Speichern von Laufzeitabhängigkeiten in einer Ebene (X-Ray SDK)

Verwenden von Powertools für AWS Lambda (Python) und AWS SAM für die Rückverfolgung

Gehen Sie wie folgt vor, um eine Hello World Python-Beispielanwendung mit integrierten Powertools for AWS Lambda (Python) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den AWS SAM. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.11
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mit der Hello World Python-Vorlage.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.11 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
```

```
    - fault count(5XX): 0
    - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

Verwenden Sie Powertools für AWS Lambda (Python) und die AWS CDK für die Rückverfolgung

Gehen Sie wie folgt vor, um eine Hello World Python-Beispielanwendung mit integrierten Powertools for AWS Lambda (Python) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den AWS CDK. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.11
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS CDK Anwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language python
```

3. Installieren Sie die Python-Abhängigkeiten.

```
pip install -r requirements.txt
```

4. Erstellen Sie ein Verzeichnis lambda_function unter dem Stammordner.

```
mkdir lambda_function
```

```
cd lambda_function
```

5. Erstellen Sie eine Datei namens app.py und fügen Sie den folgenden Code zur Datei hinzu. Dies ist der Code für die -Funktion.

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. Öffnen Sie das Verzeichnis hello_world. Sie sollten eine Datei mit dem Namen hello_world_stack.py sehen.

```
cd ..
cd hello_world
```

7. Öffnen Sie hello_world_stack.py und fügen Sie den folgenden Code in die Datei ein. Dies enthält den Lambda-Konstruktor, der die Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt, und den ApiGatewayv1-Konstruktor, der die REST-API erstellt.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_11,
```

```
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Stellen Sie die Anwendung bereit.

```
cd ..
cdk deploy
```

9. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

    Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

11. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Verwenden von ADOT zum Instrumentieren Ihrer Python-Funktionen

ADOT bietet vollständig verwaltete Lambda-Ebenen, die alles packen, was Sie zum Sammeln von Telemetriedaten mit dem OTel-SDK benötigen. Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Ebenen auch für die benutzerdefinierte Initialisierung von OTel konfigurieren. Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Python-Laufzeiten können Sie den AWS -verwaltete Lambda-Layer für ADOT Python hinzufügen, um Ihre Funktionen automatisch zu instrumentieren. Dieser Layer funktioniert sowohl für arm64- als auch für x86_64-Architekturen. Eine ausführliche Anleitung zum Hinzufügen dieser Ebene finden Sie unter AWS Distro for OpenTelemetry Lambda Support for Python in der ADOT-Dokumentation.

Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Python-Funktionen

Um Details zu Aufrufen aufzuzeichnen, die Ihre Funktion an andere Ressourcen in Ihrer Anwendung tätigt, können Sie auch den verwenden AWS X-Ray-SDK für Python. Um das SDK zu erhalten, fügen Sie das `aws-xray-sdk`-Paket den Abhängigkeiten Ihrer Anwendung hinzu.

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

In Ihrem Funktionscode können Sie AWS SDK-Clients instrumentieren, indem Sie die `boto3` Bibliothek mit dem `aws_xray_sdk.core` Modul patchen.

Example Funktion — Verfolgen eines AWS SDK-Client

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all
```

```
logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

Nachdem Sie die richtigen Abhängigkeiten hinzugefügt und die erforderlichen Codeänderungen vorgenommen haben, aktivieren Sie die Ablaufverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpretieren und X-Ray Trace

Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen X-Ray. Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle

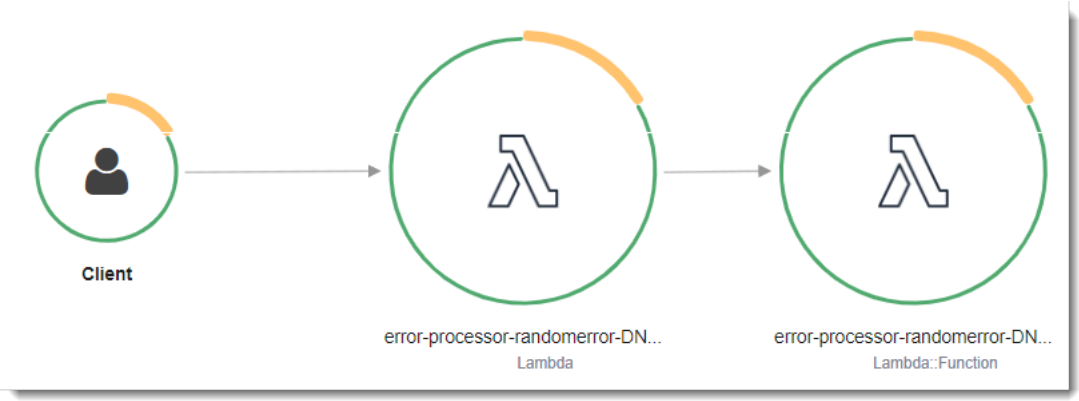
Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray Dienstdiagramm zeigt Informationen zu Ihrer Anwendung und all ihren Komponenten. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK X-Ray, um Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

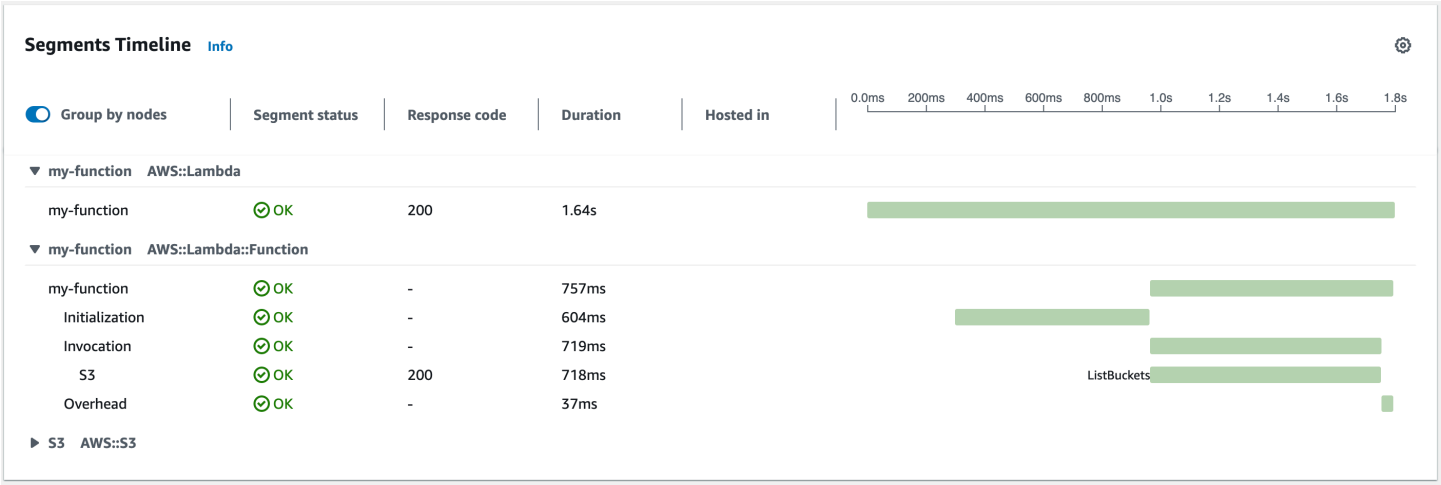


X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

In X-Ray zeichnet ein Trace Informationen über eine Anfrage auf, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können Sie geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollnachrichten und Trace-Segmenten feststellen, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.

Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray-SDK für Python im AWS X-Ray -Entwicklerhandbuch.

Preisgestaltung

Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit kostenlos nutzen. AWS Bei Überschreitung dieses Schwellenwerts X-Ray fallen Gebühren für die Speicherung und den Abruf von Trace an. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Speichern von Laufzeitabhängigkeiten in einer Ebene (X-Ray SDK)

Wenn Sie das X-Ray SDK verwenden, um AWS SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich umfangreich werden. Um zu vermeiden, dass

bei jeder Aktualisierung Ihres Funktionscodes Laufzeitabhängigkeiten hochgeladen werden, packen Sie das X-Ray SDK in eine Lambda-Schicht.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion`-Ressource, die das AWS X-Ray-SDK für Python speichert.

Example template.yml — Abhängigkeitsebene

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.11
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Änderungen, um das Layer-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Beispielanwendung blank-python .

Lambda-Ereignisfunktionen mit Ruby erstellen

Sie können Ruby-Code in AWS Lambda Event Functions ausführen. Lambda bietet Laufzeiten für Ruby, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Umgebung ausgeführt, die die Rolle AWS SDK für Ruby, with credentials from a AWS Identity and Access Management (IAM) enthält, die Sie verwalten. Weitere Informationen zu den SDK-Versionen, die in

den Laufzeiten von Ruby enthalten sind, finden Sie unter the section called “Runtime-included SDK-Versionen”.

Lambda unterstützt die folgenden Ruby-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Ruby 4.0	`ruby4.0`	Amazon Linux 2023	31. März 2029	30. April 2029	31. Mai 2029
Rubin 3.4	`ruby3.4`	Amazon Linux 2023	31. März 2028	30. April 2028	31. Mai 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	31. März 2027	29. Juli 2027	31. August 2027

Um eine Ruby-Event-Funktion zu erstellen

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Runtime: Wählen Sie Ruby 4.0.
4. Wählen Sie „Event-Funktion erstellen“.

Die Konsole erstellt eine Lambda-Ereignisfunktion mit einer einzigen Quelldatei namens`lambda_function.rb`. Mit dem integrierten Code-Editor können Sie diese Datei bearbeiten und weitere Dateien hinzufügen. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren. Um Ihren Code auszuführen, wählen Sie anschließend im Abschnitt TESTEREIGNISSE die Option Testereignis erstellen aus.

Die `lambda_function.rb`-Datei exportiert eine Funktion mit dem Namen `lambda_handler`, die ein Ereignisobjekt und ein Kontext-Objekt übernimmt. Dies ist die Handler-Funktion, die bei einem Aufruf der Funktion von Lambda aufgerufen wird. Die Laufzeit der Ruby Event Function ruft

Aufrufereignisse von Lambda ab und übergibt sie an den Handler. In der Konfiguration der Funktion lautet der Wert für den Handler `lambda_function.lambda_handler`.

Wenn Sie Ihren Funktionscode speichern, erstellt die Lambda-Konsole ein Bereitstellungspaket für das ZIP-Dateiarchiv. Wenn Sie Ihren Funktionscode außerhalb der Konsole (mit einer IDE) entwickeln, müssen Sie ein Bereitstellungspaket erstellen, um Ihren Code in die Lambda-Funktion hochzuladen.

Die Funktionslaufzeit übergibt neben dem Aufrufereignis ein Context-Objekt an den Handler. Das Context-Objekt enthält zusätzliche Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung. Weitere Informationen erhalten Sie über die Umgebungsvariablen.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- Runtime-included SDK-Versionen
- Aktivieren von Yet Another Ruby JIT (YJIT)
- Definieren Sie den Lambda-Ereignisfunktionshandler in Ruby
- Stellen Sie Ruby Lambda Event Functions mit.zip-Dateiarchiven bereit
- Stellen Sie Ruby Lambda Event Functions mit Container-Images bereit
- Arbeiten mit Ebenen für Ruby Lambda Event Functions
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Ruby-Ereignisfunktion
- Ruby Lambda Event Functions protokollieren und überwachen
- Instrumentierung von Ruby-Code in AWS Lambda

Runtime-included SDK-Versionen

Die Version des AWS SDK, die in der Ruby-Laufzeit enthalten ist, hängt von der Runtime-Version und Ihrer ab AWS-Region. Das AWS SDK for Ruby ist modular konzipiert und wird durch getrennt AWS-Service. Um die Versionsnummer eines bestimmten Service-Gems zu ermitteln, das in der von Ihnen verwendeten Runtime enthalten ist, erstellen Sie eine Lambda-Ereignisfunktion mit Code im folgenden Format. Ersetzen Sie `aws-sdk-s3` und `Aws::S3` durch den Namen der Service-Gems, die Ihr Code verwendet.

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
end
```

Aktivieren von Yet Another Ruby JIT (YJIT)

Die Ruby-Runtimes unterstützen YJIT, einen leichten, minimalistischen Ruby-JIT-Compiler. YJIT bietet eine deutlich höhere Leistung, benötigt aber auch mehr Speicher als der Ruby-Interpreter. YJIT wird für Ruby-on-Rails-Workloads empfohlen.

YJIT ist standardmäßig nicht aktiviert. Um YJIT für eine Ruby-Funktion zu aktivieren, setzen Sie die Umgebungsvariable auf. `RUBY_YJIT_ENABLE 1` Um sicherzustellen, dass YJIT aktiviert ist, drucken Sie das Ergebnis der `RubyVM::YJIT.enabled?`-Methode aus.

Example– Sicherstellen, dass YJIT aktiviert ist

```
puts(RubyVM::YJIT.enabled?())
# => true
```

Definieren Sie den Lambda-Ereignisfunktionshandler in Ruby

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Themen

- Ruby-Handler-Grundlagen
- Bewährte Codemethoden für Ruby Lambda Event Functions

Ruby-Handler-Grundlagen

Im folgenden Beispiel definiert die Datei `function.rb` eine Handler-Methode mit dem Namen `handler`. Die Handler-Funktion nimmt zwei Objekten als Eingabe und gibt ein JSON-Dokument zurück.

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

In Ihrer Funktionskonfiguration gibt die Einstellung `handler` Lambda die Stelle an, an der sich der Handler befindet. Im vorherigen Beispiel ist der korrekte Wert für diese Einstellung **`function.handler`**. Er enthält zwei Namen, die durch einen Punkt getrennt werden: den Namen der Datei und den Namen der Handler-Methode.

Sie können Ihre Handler-Methode außerdem in einer Klasse definieren. Das folgende Beispiel definiert eine Handler-Methode mit dem Namen `process` in einer Klasse mit dem Namen `Handler` in einem Modul mit dem Namen `LambdaFunctions`.

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
```

```
    end
  end
end
```

In diesem Fall ist die Handler-Einstellung **`source.LambdaFunctions::Handler.process`**.

Die zwei Objekte, die der Handler akzeptiert, sind das Aufrufereignis und Context. Das Ereignis ist ein Ruby-Objekt, das die vom Aufrufer bereitgestellte Nutzlast enthält. Wenn es sich bei der Nutzlast um ein JSON-Dokument handelt, ist das Ereignisobjekt ein Ruby-Hash. Andernfalls ist es eine Zeichenfolge. Das Context-Objekt besitzt Methoden und Eigenschaften, die Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereitstellen.

Der Funktions-Handler wird jedes Mal ausgeführt, wenn Ihre Lambda-Funktion aufgerufen wird. Statischer Code außerhalb des Handler wird einmal pro Funktions-Instance ausgeführt. Wenn Ihr Handler Ressourcen wie SDK-Clients und Datenbankverbindungen verwendet, können Sie diese außerhalb der Handler-Methode erstellen, um sie für mehrere Aufrufe wiederzuverwenden.

Jede Instance Ihrer Funktion kann zwar mehrere Aufrufereignisse verarbeiten, verarbeitet jedoch jeweils nur ein Ereignis zur gleichen Zeit. Die Anzahl der Instances, die zu einem bestimmten Zeitpunkt ein Ereignis verarbeiten, ist die Nebenläufigkeit Ihrer Funktion. Weitere Informationen zur Lambda-Ausführungsumgebung finden Sie unter Verständnis des Lebenszyklus der Lambda-Ausführungsumgebung.

Bewährte Codemethoden für Ruby Lambda Event Functions

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die Best Practices für Codierung anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Dies macht es einfacher, eine Funktion zu erstellen, die besser anhand von Einheiten getestet werden kann. In Ruby könnte das zum Beispiel so aussehen:

```
def lambda_handler(event:, context:)
    foo = event['foo']
    bar = event['bar']

    result = my_lambda_function(foo:, bar:)

end

def my_lambda_function(foo:, bar:)
```

```
    // MyLambdaFunction logic here

end
```

- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda - Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Ruby-Laufzeit beinhalten diese das AWS SDK.

  Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, aktualisiert Lambda diese Bibliotheken regelmäßig. Diese Updates können zu subtilen Änderungen am Verhalten Ihrer Lambda-Funktion führen. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch wird die Zeit reduziert, die benötigt wird, bis Ihr Bereitstellungspaket vor dem Aufruf heruntergeladen und entpackt wird. Vermeiden Sie es, bei in Ruby verfassten Funktionen die gesamte AWS SDK-Bibliothek als Teil Ihres Bereitstellungspakets hochzuladen. Verlassen Sie sich stattdessen selektiv auf die Gems, die Komponenten des benötigten SDK aufnehmen (z. B. die DynamoDB- oder Amazon S3-SDK-Gems).

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates auf die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Stellen Sie Ruby Lambda Event Functions mit.zip-Dateiarchiven bereit

Der Code Ihrer AWS Lambda Funktion besteht aus einer .rb-Datei, die den Handlercode Ihrer Funktion sowie alle zusätzlichen Abhängigkeiten (Gems) enthält, von denen Ihr Code abhängt. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Dieses Paket kann entweder ein ZIP-Dateiarchiv oder ein Container-Image sein. Weitere Informationen zur Verwendung von Container-Images mit Ruby finden Sie unter Deploy Ruby Lambda Event Functions with container images.

Zum Erstellen des Bereitstellungspakets für ein ZIP-Dateiarchiv können Sie ein integriertes Dienstprogramm für ZIP-Dateien Ihres Befehlszeilen-Tools oder ein anderes Dienstprogramm für ZIP-Dateien verwenden, wie 7zip. In den Beispiele in den folgenden Abschnitten wird davon ausgegangen, dass Sie ein `zip`-Befehlszeilen-Tool in einer Linux- oder MacOS-Umgebung verwenden. Um dieselben Befehle in Windows zu verwenden, können Sie das Windows-Subsystem für Linux installieren, um eine Windows-integrated Version von Ubuntu und Bash zu erhalten.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

In den Beispielbefehlen in den folgenden Abschnitten wird das Hilfsprogramm Bundler verwendet, um Ihrem Bereitstellungspaket Abhängigkeiten hinzuzufügen. Führen Sie zum Installieren von Bundler den folgenden Befehl aus:

```
gem install bundler
```

Sections

- Abhängigkeiten in Ruby
- ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen
- ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen
- Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten
- Erstellen von ZIP-Bereitstellungspaketen mit nativen Bibliotheken
- Ruby Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren

Abhängigkeiten in Ruby

Für Lambda-Ereignisfunktionen, die die Ruby-Laufzeit verwenden, kann eine Abhängigkeit ein beliebiges Ruby-Gem sein. Bei Bereitstellung Ihrer Funktion mithilfe eines ZIP-Archivs können Sie

diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Eine Ebene ist ein separates ZIP-Dateiarchiv, das zusätzlichen Code und andere Daten enthalten kann. Weitere Informationen zur Verwendung von Lambda-Ebenen finden Sie unter the section called “Layer”.

Die Ruby-Laufzeit beinhaltet die AWS SDK für Ruby. Wenn Ihre Funktion das SDK verwendet, müssen Sie es nicht mit Ihrem Code bündeln. Wenn Sie jedoch die volle Kontrolle über Ihre Abhängigkeiten behalten oder eine bestimmte Version des SDK verwenden möchten, können Sie es dem Bereitstellungspaket Ihrer Funktion hinzufügen.

Sie können das SDK entweder in Ihre ZIP-Datei aufnehmen oder es mithilfe einer Lambda-Ebene hinzufügen. Abhängigkeiten in Ihrer ZIP-Datei oder auf Lambda-Ebenen haben Vorrang vor Versionen, die in der Laufzeit enthalten sind. Welche Version des SDK für Ruby in Ihrer Laufzeitversion enthalten ist, erfahren Sie unter the section called “Runtime-included SDK-Versionen”.

Im Rahmen des AWS -Modells der geteilten Verantwortung sind Sie für die Verwaltung aller Abhängigkeiten in den Bereitstellungspaketen Ihrer Funktionen verantwortlich. Dies beinhaltet das Durchführen von Updates und Sicherheitspatches. Zum Aktualisieren von Abhängigkeiten im Bereitstellungspaket Ihrer Funktion erstellen Sie zunächst eine neue ZIP-Datei und laden Sie diese dann in Lambda hoch. Weitere Informationen finden Sie unter ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen und Ruby Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen

Wenn Ihr Funktionscode keine Abhängigkeiten hat, enthält Ihre .zip-Datei nur die .rb-Datei mit dem Handlercode Ihrer Funktion. Erstellen Sie mit Ihrem bevorzugten ZIP-Programm eine ZIP-Datei mit Ihrer RB-Datei im Stammverzeichnis. Wenn sich die .rb-Datei nicht im Stammverzeichnis Ihrer .zip-Datei befindet, kann Lambda Ihren Code nicht ausführen.

Informationen zum Bereitstellen Ihrer ZIP-Datei zum Erstellen einer neuen Lambda-Funktion oder Aktualisieren einer vorhandenen Funktion, finden Sie unter Ruby Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen

Hängt Ihr Funktionscode von zusätzlichen Ruby-Gems ab, können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode in Ihre ZIP-Datei aufnehmen oder eine Lambda-

Ebene verwenden. In diesem Abschnitt erfahren Sie, wie Sie Ihre Abhängigkeiten in Ihr ZIP-Bereitstellungspaket aufnehmen. Anweisungen zum Einschließen Ihrer Abhängigkeiten in eine Ebene finden Sie unter the section called “Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten”.

Angenommen, Ihr Funktionscode ist in einer Datei mit dem Namen `lambda_function.rb` in Ihrem Projektverzeichnis gespeichert. Die folgenden CLI-Befehle erstellen Sie eine ZIP-Datei mit dem Namen „`my_deployment_package.zip`“, die Ihren Funktionscode und seine Abhängigkeiten enthält.

Erstellen des Bereitstellungspakets

1. Erstellen Sie in Ihrem Projektverzeichnis eine `Gemfile`-Datei für die Angabe Ihrer Abhängigkeiten.

   ```
   bundle init
   ```

2. Bearbeiten Sie die `Gemfile`-Datei mit Ihrem bevorzugten Texteditor, um die Abhängigkeiten Ihrer Funktion anzugeben. Wenn Sie also beispielsweise das TZInfo-Gem verwenden möchten, bearbeiten Sie Ihre `Gemfile`-Datei so, dass sie wie folgt aussieht:

   ```
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. Führen Sie den folgenden Befehl aus, um die in Ihrer `Gemfile`-Datei angegebenen Gems in Ihrem Projektverzeichnis zu installieren. Durch diesen Befehl wird `vendor/bundle` als Standardpfad für Gem-Installationen festgelegt:

   ```
   bundle config set --local path 'vendor/bundle' && bundle install
   ```

   Die Ausgabe sollte in etwa wie folgt aussehen:

   ```
   Fetching gem metadata from https://rubygems.org/...........
   Resolving dependencies...
   Using bundler 2.4.13
   Fetching tzinfo 2.0.6
   Installing tzinfo 2.0.6
   ...
   ```

> **Note**
>
> Wenn Sie Gems später wieder global installieren möchten, können Sie den folgenden Befehl ausführen:
>
> ```
> bundle config set --local system 'true'
> ```

4. Erstellen Sie ein ZIP-Dateiarchiv, das die `lambda_function.rb`-Datei mit dem Handler-Code Ihrer Funktion sowie mit den Abhängigkeiten enthält, die Sie im vorherigen Schritt installiert haben.

   ```
   zip -r my_deployment_package.zip lambda_function.rb vendor
   ```

   Die Ausgabe sollte in etwa wie folgt aussehen:

   ```
   adding: lambda_function.rb (deflated 37%)
     adding: vendor/ (stored 0%)
     adding: vendor/bundle/ (stored 0%)
     adding: vendor/bundle/ruby/ (stored 0%)
     adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
     adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
     adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
     adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
   ...
   ```

Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten

Informationen zum Verpacken Ihrer Ruby-Abhängigkeiten in eine Lambda-Ebene finden Sie unter the section called “Layer”.

Erstellen von ZIP-Bereitstellungspaketen mit nativen Bibliotheken

Viele gängige Ruby-Gems wie `nokogirinio4r`, und `mysql` enthalten native, in C geschriebene Erweiterungen. Wenn Sie Ihrem Bereitstellungspaket Bibliotheken hinzufügen, die C-Code enthalten, müssen Sie Ihr Paket korrekt erstellen, um sicherzustellen, dass es mit der Lambda-Ausführungsumgebung kompatibel ist.

Für Produktionsanwendungen empfehlen wir, Ihren Code mithilfe von AWS Serverless Application Model (AWS SAM) zu erstellen und bereitzustellen. AWS SAM Verwenden Sie die `sam build --use-container` Option, um Ihre Funktion in einem Lambda-like Docker-Container zu erstellen. Weitere Informationen AWS SAM zur Bereitstellung Ihres Funktionscodes finden Sie im AWS SAM Entwicklerhandbuch unter Anwendungen erstellen.

Um ein .zip-Bereitstellungspaket zu erstellen, das Gems mit nativen Erweiterungen enthält AWS SAM, ohne es zu verwenden, können Sie alternativ einen Container verwenden, um Ihre Abhängigkeiten in einer Umgebung zu bündeln, die der Lambda Ruby-Laufzeitumgebung entspricht. Für diese Schritte muss Docker auf Ihrem Build-Computer installiert sein. Weitere Informationen zur Installation von Docker finden Sie unter Install Docker Engine.

So erstellen Sie ein ZIP-Bereitstellungspaket in einem Docker-Container

1. Erstellen Sie auf Ihrem lokalen Build-Computer einen Ordner als Speicherort für Ihren Container. Erstellen Sie in diesem Ordner eine Datei mit dem Namen `dockerfile` und fügen Sie folgenden Code in die Datei ein:

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. Führen Sie in dem Ordner, in dem Sie Ihre `dockerfile`-Datei erstellt haben, den folgenden Befehl aus, um den Docker-Container zu erstellen:

   ```
   docker build -t awsruby32 .
   ```

3. Navigieren Sie zu dem Projektverzeichnis, das die Datei vom Typ `.rb` mit dem Handler-Code Ihrer Funktion sowie die `Gemfile`-Datei mit den Abhängigkeiten Ihrer enthält. Führen Sie in diesem Verzeichnis den folgenden Befehl aus, um den Lambda-Ruby-Container zu starten:

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

> **Note**
>
> In macOS wird möglicherweise eine Warnung mit dem Hinweis angezeigt, dass die Plattform des angeforderten Images nicht der erkannten Hostplattform entspricht. Ignorieren Sie diese Warnung.

Windows PowerShell

```
docker run --rm -it -v ${pwd}:/var/task -w /var/task awsruby32
```

Wenn Ihr Container startet, sollte eine Bash-Aufforderung angezeigt werden.

```
bash-4.2#
```

4. Konfigurieren Sie das Hilfsprogramm „bundle“, um die in Ihrer `Gemfile`-Datei angegebenen Gems in einem lokalen Verzeichnis namens `vendor/bundle` zu installieren und Ihre Abhängigkeiten zu installieren.

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. Erstellen Sie das ZIP-Bereitstellungspaket mit Ihrem Funktionscode und den zugehörigen Abhängigkeiten. In diesem Beispiel hat die Datei, die den Handler-Code Ihrer Funktion enthält, den Namen `lambda_function.rb`.

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. Verlassen Sie den Container und kehren Sie zu Ihrem lokalen Projektverzeichnis zurück.

```
bash-4.2# exit
```

Nun können Sie Ihre Lambda-Funktion mithilfe des ZIP-Bereitstellungspakets erstellen oder aktualisieren. Siehe Ruby Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren

Ruby Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Sie können Ihr .zip-Paket mithilfe der Lambda-Konsole, der und der AWS Command Line Interface Lambda-API bereitstellen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Create Event Function aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.
   c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.
4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie „Event-Funktion erstellen“. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.

3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:

    a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.
    b. Klicken Sie auf Open.
    c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

ZIP-Dateifunktionen mithilfe des Konsolencode-Editors aktualisieren

Für einige Funktionen mit ZIP-Bereitstellungspaketen können Sie Ihren Funktionscode direkt mit dem in der Lambda-Konsole integrierten Code-Editor aktualisieren. Zur Verwendung dieses Features muss Ihre Funktion folgende Kriterien erfüllen:

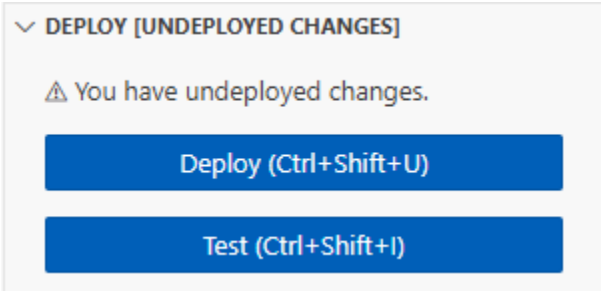
- Ihre Funktion muss eine der interpretierten Sprachlaufzeiten verwenden (Python oder Ruby) Node.js
- Das Bereitstellungspaket Ihrer Funktion muss kleiner als 50 MB (entpackt) sein.

Funktionscode für Funktionen mit Container-Image-Bereitstellungspaketen kann nicht direkt in der Konsole bearbeitet werden.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.

4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Befehle Funktion erstellen und Funktionscode aktualisieren, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre ZIP-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip-Datei mit dem zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten erstellt. Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Sie können es auch verwenden AWS SAM , um eine Lambda-Funktion mithilfe eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion zu erstellen AWS SAM, können Sie Ihre ZIP-Datei in einem Amazon S3 S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer AWS SAM Vorlage spezifiziert die `AWS::Serverless::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri`- auf die Amazon S3 S3-URI, den Pfad zum lokalen Ordner oder FunctionCodeObjekt des Funktionscodes gesetzt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Wenn Ihre ZIP-Datei größer als 50 MB ist, müssen Sie sie nicht zuerst in einen Amazon S3 S3-Bucket hochladen. AWS SAM AWS SAM kann .zip-Pakete bis zur maximal zulässigen Größe von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei in finden Sie unter AWS::Serverless: AWS SAM:Function im Entwicklerhandbuch.AWS SAM

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit CloudFormation

Sie können verwenden CloudFormation , um eine Lambda-Funktion mithilfe eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in

einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die CloudFormation generierte ZIP-Datei darf 4 MB nicht überschreiten. Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei finden Sie unter AWS::Lambda: :Function im Benutzerhandbuch. CloudFormationCloudFormation

Stellen Sie Ruby Lambda Event Functions mit Container-Images bereit

Es gibt drei Möglichkeiten, ein Container-Image für eine Ruby Lambda Event Function zu erstellen:

- Mit einem AWS Basisimage für Ruby

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Mit einem AWS OS-only Basisbild

  AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ruby in das Image aufnehmen.

- Verwenden Sie ein AWS Nicht-Basis-Image

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ruby in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für Ruby
- Mit einem AWS Basisimage für Ruby

- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für Ruby

AWS stellt die folgenden Basis-Images für Ruby bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
4,0	Ruby 4.0	Amazon Linux 2023	Dockerfile für Ruby 4.0 auf GitHub	31. März 2029
3.4	Rubin 3.4	Amazon Linux 2023	Dockerfile für Ruby 3.4 auf GitHub	31. März 2028
3.3	Ruby 3.3	Amazon Linux 2023	Dockerfile für Ruby 3.3 auf GitHub	31. März 2027

Amazon ECR-Repository: gallery.ecr. aws/lambda/ruby

Mit einem AWS Basisimage für Ruby

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Ruby

Erstellen eines Images aus einem Base Image

Erstellen eines Container-Images für Ruby

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir example
cd example
```

2. Erstellen Sie eine neue Datei mit dem Namen `Gemfile`. Hier listen Sie die benötigten RubyGems Pakete für Ihre Anwendung auf. Das AWS SDK für Ruby ist erhältlich bei RubyGems. Sie sollten bestimmte AWS Service Gems für die Installation auswählen. Um beispielsweise das Ruby-Gem für Lambda zu verwenden, sollte Ihr Gemfile so aussehen:

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

   Alternativ enthält das aws-sdk-Gem jedes verfügbare AWS Service-Gem. Dieses Gem ist sehr groß. Wir empfehlen, es nur zu verwenden, wenn Sie auf viele AWS Dienste angewiesen sind.

3. Installieren Sie die im Gemfile angegebenen Abhängigkeiten mithilfe von bundle install.

```
bundle install
```

4. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.rb`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Ruby-Funktion

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Erstellen Sie eine neue Docker-Datei. Es folgt ein Beispiel für ein Dockerfile, das ein AWS - Basis-Image verwendet. Dieses Dockerfile verwendet die folgende Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in eine Lambda-defined Umgebungsvariable zu `{LAMBDA_TASK_ROOT}` kopieren.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.4

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den

Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.
   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-only Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Runtime-Interface-Client in Ihr Image aufnehmen. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Lambda Runtime Interface Client für Ruby mit dem RubyGems.org Paketmanager:

```
gem install aws_lambda_ric
```

Sie können den Ruby-Runtime-Interface-Client auch von GitHub herunterladen.

Das folgende Beispiel zeigt, wie ein Container-Image für Ruby mithilfe eines AWS Nicht-Basis-Images erstellt wird. Das Beispiel-Dockerfile verwendet ein offizielles Ruby-Basis-Image. Das Docker-File enthält den Laufzeitschnittstellen-Client.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Ruby

Erstellen eines Images aus einem alternativen Basis-Image

Erstellen eines Container-Images für Ruby mit einem alternativen Basis-Image

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `Gemfile`. Hier listen Sie die benötigten RubyGems Pakete für Ihre Anwendung auf. Das AWS SDK für Ruby ist erhältlich bei RubyGems. Sie sollten bestimmte AWS Service Gems für die Installation auswählen. Um beispielsweise das Ruby-Gem für Lambda zu verwenden, sollte Ihr Gemfile so aussehen:

   ```
   source 'https://rubygems.org'

   gem 'aws-sdk-lambda'
   ```

   Alternativ enthält das aws-sdk-Gem jedes verfügbare AWS Service-Gem. Dieses Gem ist sehr groß. Wir empfehlen, es nur zu verwenden, wenn Sie auf viele AWS Dienste angewiesen sind.

3. Installieren Sie die im Gemfile angegebenen Abhängigkeiten mithilfe von bundle install.

   ```
   bundle install
   ```

4. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.rb`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Ruby-Funktion

   ```
   module LambdaFunction
   ```

```
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Erstellen Sie eine neue Docker-Datei. Das folgende Dockerfile verwendet ein Ruby-Basis-Image anstelle eines AWS -Basis-Images. Das Dockerfile enthält den Laufzeitschnittstellen-Client für Ruby, der das Image mit Lambda kompatibel macht. Alternativ können Sie den Laufzeitschnittstellen-Client zum Gemfile Ihrer Anwendung hinzufügen.

   - Legen Sie die `FROM`-Eigenschaft auf das Ruby-Basis-Image fest.
   - Erstellen Sie ein Verzeichnis für den Funktionscode und eine Umgebungsvariable, die auf dieses Verzeichnis verweist. In diesem Beispiel ist das Verzeichnis `/var/task`, das die Lambda-Ausführungsumgebung widerspiegelt. Sie können jedoch ein beliebiges Verzeichnis für den Funktionscode wählen, da das Dockerfile kein AWS Basis-Image verwendet.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   Example Dockerfile

```
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
```

```
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

Linux/macOS

Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

   Wenn erfolgreich, sehen Sie eine Antwort wie diese:

   ```
   {
       "repository": {
           "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
   world",
           "registryId": "111122223333",
           "repositoryName": "hello-world",
           "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world",
           "createdAt": "2023-03-09T10:39:01+00:00",
           "imageTagMutability": "MUTABLE",
           "imageScanningConfiguration": {
               "scanOnPush": true
   ```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Ruby Lambda Event Functions

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Verwenden von Gems aus Ebenen in einer Funktion
- Fügen Sie die Ebene zu Ihrer Funktion hinzu
- Beispiel-App

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Erstellen Sie die Ebene mithilfe derselben Version von Ruby, die Sie für die Lambda-Funktion verwenden möchten. Wenn Sie beispielsweise Ihre Ebene für Ruby 4.0 erstellen, verwenden Sie die Ruby 4.0-Laufzeit für Ihre Funktion.
- Die ZIP-Datei Ihrer Ebene muss eine der folgenden Verzeichnisstrukturen verwenden:
  - `ruby/gems/`*`x.x.x`*(wo *`x.x.x`* ist zum Beispiel Ihre Ruby-Version`3.4.0`)
  - `ruby/lib`

  Weitere Informationen finden Sie unter Ebenenpfade für jede Lambda-Laufzeit.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder Ruby-Gems von Drittanbietern oder Ihre eigenen Ruby-Module und -Klassen enthalten. Viele beliebte Ruby-Gems enthalten native Erweiterungen (C-Code), die für die Lambda-Linux-Umgebung kompiliert werden müssen.

Reine Ruby-Gems

Reine Ruby-Gems enthalten nur Ruby-Code und müssen nicht kompiliert werden. Diese Gems lassen sich einfacher verpacken und funktionieren plattformübergreifend.

So erstellen Sie eine Ebene mit reinen Ruby-Gems

1. Erstellen Sie eine `Gemfile`, um die reinen Ruby-Gems anzugeben, die Sie in Ihre Ebene aufnehmen möchten:

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'tzinfo'
   ```

2. Installieren Sie die Gems mit Bundler in einem `vendor/bundle`-Verzeichnis:

   ```
   bundle config set --local path vendor/bundle
   bundle install
   ```

3. Kopieren Sie die installierten Gems in die Verzeichnisstruktur, die Lambda benötigt (`ruby/gems/3.4.0`):

   ```
   mkdir -p ruby/gems/3.4.0
   cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/
   ```

4. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### concurrent-ruby-1.3.5/
        #   ### tzinfo-2.0.6/
        ### specifications/
        ### cache/
        ### build_info/
        ### (other bundler directories)
```

> **Note**
>
> Sie müssen jede Gem einzeln in Ihrem Funktionscode als erforderlich festlegen. Sie können nicht `bundler/setup` oder `Bundler.require` verwenden. Weitere Informationen finden Sie unter Verwenden von Gems aus Ebenen in einer Funktion.

Gems mit nativen Erweiterungen

Viele beliebte Ruby-Gems enthalten native Erweiterungen (C-Code), die für die Zielplattform kompiliert werden müssen. Beliebte Gems mit nativen Erweiterungen sind unter anderem nokogiri, pg, mysql2, sqlite3 und ffi. Diese Gems müssen in einer Linux-Umgebung erstellt werden, die mit der Lambda-Laufzeit kompatibel ist.

So erstellen Sie eine Ebene mit Gems mit nativen Erweiterungen

1. Erstellen Sie einen `Gemfile`.

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'nokogiri'
   gem 'httparty'
   ```

2. Verwenden Sie Docker, um die Gems in einer Linux-Umgebung zu erstellen, die mit Lambda kompatibel ist. Geben Sie ein AWS -Basisabbild in Ihrem Dockerfile an:

   Example Dockerfile für Ruby 4.0

```
FROM public.ecr.aws/lambda/ruby:3.4

# Copy Gemfile
COPY Gemfile ./

# Install system dependencies for native extensions
RUN dnf update -y && \
    dnf install -y gcc gcc-c++ make

# Configure bundler and install gems
RUN bundle config set --local path vendor/bundle && \
    bundle install

# Create the layer structure
RUN mkdir -p ruby/gems/3.4.0 && \
    cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/

# Create the layer zip file
RUN zip -r layer.zip ruby/
```

3. Erstellen Sie das Bild und extrahieren Sie die Ebene:

```
docker build -t ruby-layer-builder .
docker run --rm -v $(pwd):/output --entrypoint cp ruby-layer-builder layer.zip /
output/
```

   Dadurch werden die Gems in der richtigen Linux-Umgebung erstellt und die `layer.zip`-Datei in Ihr lokales Verzeichnis kopiert. Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### bigdecimal-3.2.2/
        #   ### csv-3.3.5/
        #   ### httparty-0.23.1/
        #   ### mini_mime-1.1.5/
```

```
        #   ### multi_xml-0.7.2/
        #   ### nokogiri-1.18.8-x86_64-linux-gnu/
        #   ### racc-1.8.1/
        ### build_info/
        ### cache/
        ### specifications/
        ### (other bundler directories)
```

> **Note**
>
> Sie müssen jede Gem einzeln in Ihrem Funktionscode als erforderlich festlegen. Sie können nicht `bundler/setup` oder `Bundler.require` verwenden. Weitere Informationen finden Sie unter Verwenden von Gems aus Ebenen in einer Funktion.

Benutzerdefinierte Ruby-Module

So erstellen Sie eine Ebene mithilfe Ihres eigenen Codes

1. Erstellen Sie die erforderliche Verzeichnisstruktur für Ihre Ebene:

   ```
   mkdir -p ruby/lib
   ```

2. Erstellen Sie Ihre Ruby-Module im `ruby/lib`-Verzeichnis. Das folgende Beispielmodul validiert Bestellungen, indem es überprüft, ob sie die erforderlichen Informationen enthalten.

   Example ruby/lib/order_validator.rb

   ```
   require 'json'

   module OrderValidator
     class ValidationError < StandardError; end

     def self.validate_order(order_data)
       # Validates an order and returns formatted data
       required_fields = %w[product_id quantity]

       # Check required fields
       missing_fields = required_fields.reject { |field| order_data.key?(field) }
       unless missing_fields.empty?
   ```

```
      raise ValidationError, "Missing required fields: #{missing_fields.join(',
 ')}"
    end

    # Validate quantity
    quantity = order_data['quantity']
    unless quantity.is_a?(Integer) && quantity > 0
      raise ValidationError, 'Quantity must be a positive integer'
    end

    # Format and return the validated data
    {
      'product_id' => order_data['product_id'].to_s,
      'quantity' => quantity,
      'shipping_priority' => order_data.fetch('priority', 'standard')
    }
  end

  def self.format_response(status_code, body)
    # Formats the API response
    {
      statusCode: status_code,
      body: JSON.generate(body)
    }
  end
end
```

3. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

   ```
   ruby/
   ### lib/
   ```

```
    ### order_validator.rb
```

4. Fordern Sie in Ihrer Funktion die Module an und verwenden Sie sie. Sie müssen jede Gem einzeln in Ihrem Funktionscode als erforderlich festlegen. Sie können nicht `bundler/setup` oder `Bundler.require` verwenden. Weitere Informationen finden Sie unter Verwenden von Gems aus Ebenen in einer Funktion. Beispiel:

```
require 'json'
require 'order_validator'

def lambda_handler(event:, context:)
  begin
    # Parse the order data from the event body
    order_data = JSON.parse(event['body'] || '{}')

    # Validate and format the order
    validated_order = OrderValidator.validate_order(order_data)

    OrderValidator.format_response(200, {
      message: 'Order validated successfully',
      order: validated_order
    })
  rescue OrderValidator::ValidationError => e
    OrderValidator.format_response(400, {
      error: e.message
    })
  rescue => e
    OrderValidator.format_response(500, {
      error: 'Internal server error'
    })
  end
end
```

Sie können das folgende Testereignis verwenden, um die Funktion aufzurufen:

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Erwartete Antwort:

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"product_id\":
\"ABC123\",\"quantity\":2,\"shipping_priority\":\"express\"}}"
}
```

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den AWS CLI Befehl publish-layer-version aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes ruby4.0
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.
4. (Optional) Wählen Sie für Kompatible Laufzeiten die Ruby-Laufzeit aus, die der Ruby-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Verwenden von Gems aus Ebenen in einer Funktion

In Ihrem Funktionscode müssen Sie jedes Gem, das Sie verwenden möchten, explizit als erforderlich festlegen. Bundler-Befehle wie `bundler/setup` und `Bundler.require` werden nicht unterstützt. So verwenden Sie Gems aus einer Ebene in einer Lambda-Funktion:

```
# Correct: Use explicit requires for each gem
require 'nokogiri'
require 'httparty'

def lambda_handler(event:, context:)
  # Use the gems directly
  doc = Nokogiri::HTML(event['html'])
  response = HTTParty.get(event['url'])
  # ... rest of your function
end

# Incorrect: These Bundler commands will not work
# require 'bundler/setup'
# Bundler.require
```

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Um die Ebene an Ihre Funktion anzuhängen, führen Sie den Befehl update-function-configuration aus. AWS CLI Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.

4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

> **Note**
>
> Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Beispiel-App

Weitere Beispiele für die Verwendung von Lambda-Layern finden Sie in der Layer-Ruby-Beispielanwendung im AWS Lambda Developer Guide Repository. GitHub Diese Anwendung enthält eine Ebene, die die tzinfo-Bibliothek enthält. Nach dem Erstellen der Ebene können Sie die entsprechende Funktion bereitstellen und aufrufen, um zu bestätigen, dass die Ebene wie erwartet funktioniert.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Ruby-Ereignisfunktion

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Methoden

- `get_remaining_time_in_millis` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `function_name` – Der Name der Lambda-Funktion.
- `function_version` – Die Version der Funktion.
- `invoked_function_arn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memory_limit_in_mb` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `aws_request_id` – Der Bezeichner der Aufrufanforderung.
- `log_group_name` – Protokollgruppe für die Funktion.
- `log_stream_name` – Der Protokollstrom für die Funktionsinstance.
- `deadline_ms` – Das Datum, an dem eine Zeitüberschreitung für die Ausführung eintritt (in Unix-Millisekunden).
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `client_context` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.

Ruby Lambda Event Functions protokollieren und überwachen

AWS Lambda überwacht Lambda Event Functions automatisch in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Funktion umfasst eine CloudWatch Logs-Protokollgruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Lambda Event Function-Logs an CloudWatch Logs senden.

Auf dieser Seite wird beschrieben, wie Sie Protokollausgaben aus dem Code Ihrer Lambda-Funktion erstellen und mit der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Erweiterte Lambda-Logging-Kontrollen mit Ruby verwenden
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)
- Löschen von Protokollen
- Arbeiten mit der Ruby-Logger-Bibliothek

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie `puts`-Anweisungen verwenden oder eine Protokollierungsbibliothek, die zu `stdout` oder `stderr` schreibt. Das folgende Beispiel protokolliert die Werte der Umgebungsvariablen und dem Ereignisobjekt. Informationen über das Kontextobjekt, das Ihrem Handler zur Verfügung steht, finden Sie unter. the section called “Kontext”

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
```

```
    puts event.to_a
end
```

Example Protokollformat

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Die Ruby-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Für detailliertere Protokolle verwenden Sie die the section called “Arbeiten mit der Ruby-Logger-Bibliothek”.

Erweiterte Lambda-Logging-Kontrollen mit Ruby verwenden

Um Ihnen mehr Kontrolle darüber zu geben, wie die Logs Ihrer Funktionen erfasst, verarbeitet und genutzt werden, bietet Lambda erweiterte Logging-Kontrollen mit Ruby. Für Laufzeiten von Ruby 4.0 und höher können Sie die folgenden Protokollierungsoptionen konfigurieren:

- Protokollformat — wählen Sie zwischen Klartext und strukturiertem JSON-Format für die Logs Ihrer Funktion
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, die Lambda an Amazon sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Protokolle sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.

Weitere Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Ruby-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Strukturierte JSON-Logs mit Ruby verwenden

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die von der `Logger` Ruby-Standardbibliothek ausgegebenen Protokolle CloudWatch als strukturiertes JSON an. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Die `Logger` Ruby-Bibliothek kann auch zusätzliche Schlüsselwertpaare hinzufügen, z. B. `"logger"` zu diesem JSON-Objekt.

Die Beispiele in den folgenden Abschnitten zeigen, wie mit der `Logger` Ruby-Bibliothek generierte Protokollausgaben in CloudWatch Logs erfasst werden, wenn Sie das Protokollformat Ihrer Funktion als JSON konfigurieren.

Beachten Sie, dass Lambda, wenn Sie die `puts` Methode verwenden, um grundlegende Protokollausgaben zu erzeugenthe section called “Erstellen einer Funktion, die Protokolle zurückgibt”, wie unter beschrieben, diese Ausgaben als Klartext erfasst, auch wenn Sie das Protokollierungsformat Ihrer Funktion als JSON konfigurieren.

Standard-JSON-Protokollausgaben mithilfe der Ruby Logger-Bibliothek

Der folgende Beispielcodeausschnitt und die Protokollausgabe zeigen, wie mit der `Logger` Ruby-Bibliothek generierte Standardprotokollausgaben in CloudWatch Logs erfasst werden, wenn das Protokollformat Ihrer Funktion auf JSON gesetzt ist.

Example Beispiel für einen Ruby-Logging-Code

```
require 'logger'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info("Inside the handler function")
end
```

Example Beispiel für einen JSON-Protokolleintrag

```
{
    "timestamp": "2025-10-27T19:17:45.586Z",
    "level": "INFO",
    "message": "Inside the handler function",
    "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Protokollieren zusätzlicher Parameter in JSON

Wenn das Protokollformat Ihrer Funktion auf JSON gesetzt ist, können Sie auch zusätzliche Parameter mit der `Logger` Ruby-Bibliothek protokollieren, indem Sie einen Hash mit zusätzlichen Schlüsselwertpaaren an die Protokollausgabe übergeben.

Example Beispiel für einen Ruby-Logging-Code

```
require 'logger'
```

```
require 'json'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  extra_params = { "a" => "b", "b" => [3] }
  logger.info({ message: "extra parameters example" }.merge(extra_params).to_json)
end
```

Example Beispiel für einen JSON-Protokolleintrag

```
{
    "timestamp": "2025-11-02T15:26:28Z",
    "level": "INFO",
    "message": "extra parameters example",
    "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
    "a": "b",
    "b": [
        3
    ]
}
```

Protokollierungsausnahmen in JSON

Der folgende Codeausschnitt zeigt, wie Ruby-Ausnahmen in der Protokollausgabe Ihrer Funktion erfasst werden, wenn Sie das Protokollformat als JSON konfigurieren. Beachten Sie, dass Protokollausgaben, die `logger.error` mit einer Ausnahme generiert wurden, der Log-Level ERROR zugewiesen wird.

Example Beispiel für einen Ruby-Logging-Code

```
require 'logger'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  begin
    raise "exception"
  rescue => e
    logger.error(e)
  end
end
```

Example Beispiel für einen JSON-Protokolleintrag

```
{
    "timestamp": "2025-11-02T16:18:57Z",
    "level": "ERROR",
    "message": "exception",
    "stackTrace": [
        "  /var/task/lambda_function.rb:4:in `lambda_handler'"
    ],
    "errorType": "RuntimeError",
    "errorMessage": "exception",
    "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
    "location": "/var/task/lambda_function.rb:lambda_handler:6"
}
```

Verwendung der Filterung auf Protokollebene mit Ruby

Durch die Konfiguration der Filterung auf Protokollebene können Sie festlegen, dass nur Protokolle mit einer bestimmten Protokollebene oder niedriger an Logs gesendet werden. CloudWatch Informationen zur Konfiguration der Filterung auf Protokollebene für Ihre Funktion finden Sie unter the section called “Log-level filtern”.

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mithilfe der `Logger` Ruby-Standardbibliothek und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. AWS Lambda filtert dann Ihre Protokollausgaben anhand des `"level"` Schlüssel-Wert-Paars im JSON-Objekt, wie unter beschriebenthe section called “Strukturierte JSON-Logs mit Ruby verwenden”. Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unterthe section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.
- Verwenden Sie eine andere Logging-Bibliothek oder Methode, um strukturierte JSON-Logs in Ihrem Code zu erstellen, die ein `"level"` Schlüssel-Wertepaar enthalten, das die Ebene der Protokollausgabe definiert.

Sie können auch eine `puts` Anweisung verwenden, um ein JSON-Objekt auszugeben, das eine Kennung auf Protokollebene enthält. Die folgende `puts` Anweisung erzeugt eine Ausgabe im JSON-Format, bei der die Protokollebene auf INFO gesetzt ist. AWS Lambda sendet das JSON-Objekt an

CloudWatch Logs, wenn die Protokollierungsebene Ihrer Funktion auf INFO, DEBUG oder TRACE gesetzt ist.

```
puts '{"msg":"My log message", "level":"info"}'
```

Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Verwenden Sie eine alternative Logging-Bibliothek

Wenn Sie eine benutzerdefinierte Version der `logger` Bibliothek verwenden müssen, können Sie sie in Ihr Bereitstellungspaket oder in eine Lambda-Schicht aufnehmen und die `RUBYLIB` Umgebungsvariable auf das `lib` Verzeichnis der Bibliothek setzen. Die Lambda-Laufzeit lädt Ihre Version anstelle der gebündelten.

Wenn Ihr Code bereits eine andere Logging-Bibliothek verwendet, um strukturierte JSON-Logs zu erstellen, müssen Sie keine Änderungen vornehmen. AWS Lambda codiert keine Protokolle doppelt, die bereits JSON-kodiert sind. Selbst wenn Sie Ihre Funktion so konfigurieren, dass sie das JSON-Protokollformat verwendet, erscheinen Ihre Logging-Ausgaben CloudWatch in der von Ihnen definierten JSON-Struktur.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.

2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
```

```
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Arbeiten mit der Ruby-Logger-Bibliothek

Die Ruby-Logger-Bibliothek gibt optimierte Protokolle zurück, die leicht zu lesen sind. Verwenden Sie das Logger-Serviceprogramm, um detaillierte Informationen, Meldungen und Fehlercodes im Zusammenhang mit Ihrer Funktion auszugeben.

```
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
```

```
  event.to_a
end
```

Die Ausgabe aus `logger` umfasst die Protokollebene, den Zeitstempel und die Anforderungs-ID.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

Instrumentierung von Ruby-Code in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , damit Sie Lambda-Anwendungen verfolgen, debuggen und optimieren können. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, von der Frontend-API bis hin zu Speicher und Datenbank im Backend. Indem Sie die X-Ray SDK-Bibliothek einfach zu Ihrer Build-Konfiguration hinzufügen, können Sie Fehler und Latenz für jeden Aufruf aufzeichnen, den Ihre Funktion an einen Dienst tätigt. AWS

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray Service-Diagramm zeigt Informationen zu Ihrer Anwendung und all ihren Komponenten. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK X-Ray, um Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.



Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.

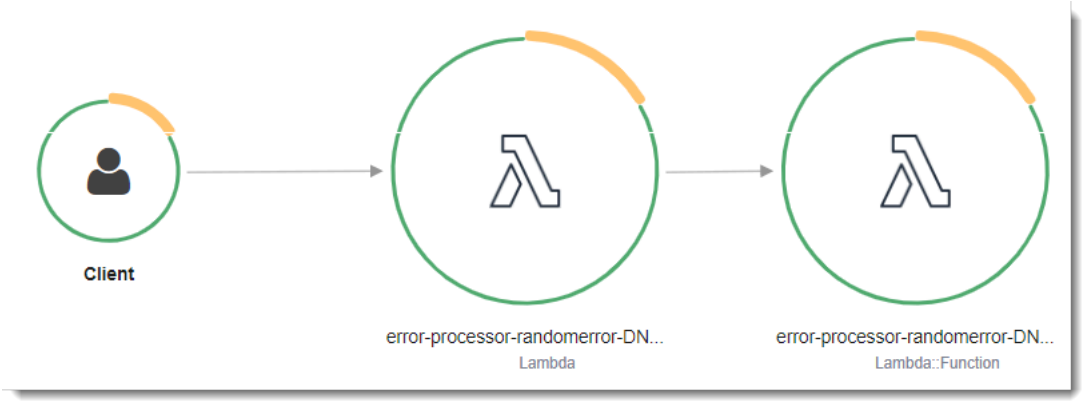
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

> **Preisgestaltung**
>
> Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit AWS kostenlos nutzen. Bei Überschreitung dieses Schwellenwerts X-Ray fallen Gebühren für die Speicherung und den Abruf von Trace an. Weitere Informationen finden Sie unter AWS X-Ray Preise.

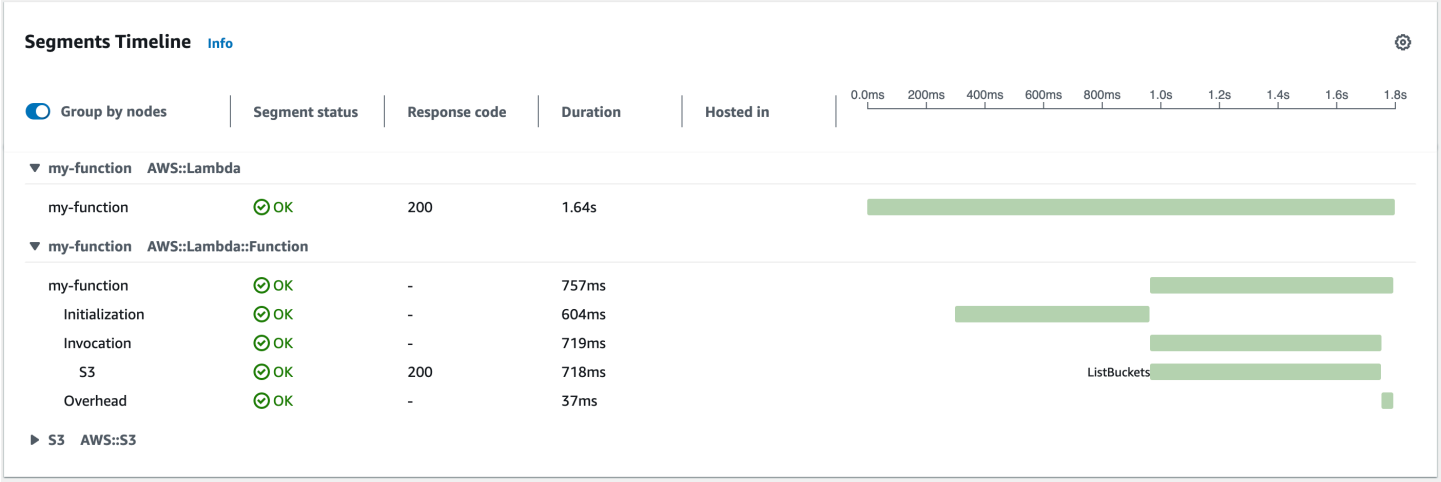
Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen. X-Ray Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

X-Ray verfolgt nicht alle Anfragen zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

In X-Ray zeichnet ein Trace Informationen über eine Anfrage auf, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen stellen Sie möglicherweise geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollmeldungen und Trace-Segmenten fest, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Sie können Ihren Handler-Code instrumentieren, um Metadaten aufzuzeichnen und Downstream-Aufrufe nachzuverfolgen. Verwenden Sie das X-Ray SDK for Ruby, um Details zu Aufrufen aufzuzeichnen, die Ihr Handler an andere Ressourcen und Dienste tätigt. Um das SDK zu erhalten, fügen Sie das `aws-xray-sdk`-Paket den Abhängigkeiten Ihrer Anwendung hinzu.

Example leer- ruby/function /Gemfile

```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

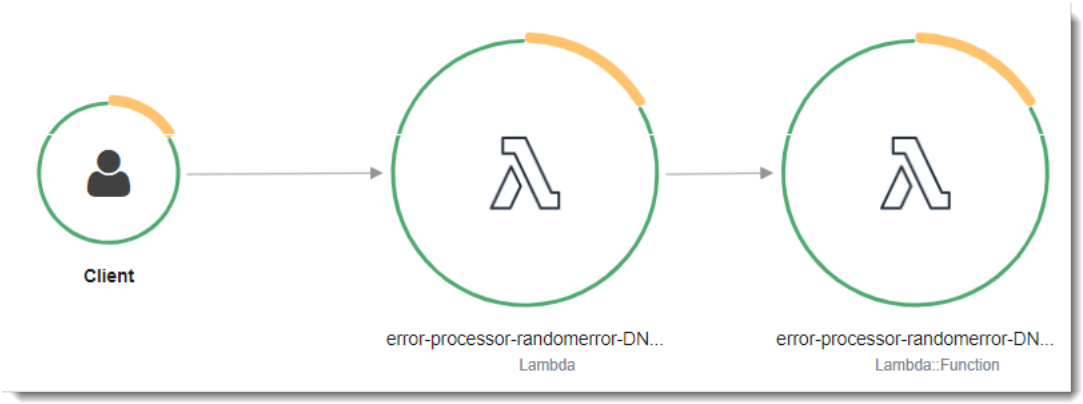
Um AWS SDK-Clients zu instrumentieren, benötigen Sie das `aws-xray-sdk/lambda` Modul, nachdem Sie einen Client im Initialisierungscode erstellt haben.

Example blank- /lambda_function.rb ruby/function — Auf der Suche nach einem AWS SDK-Client

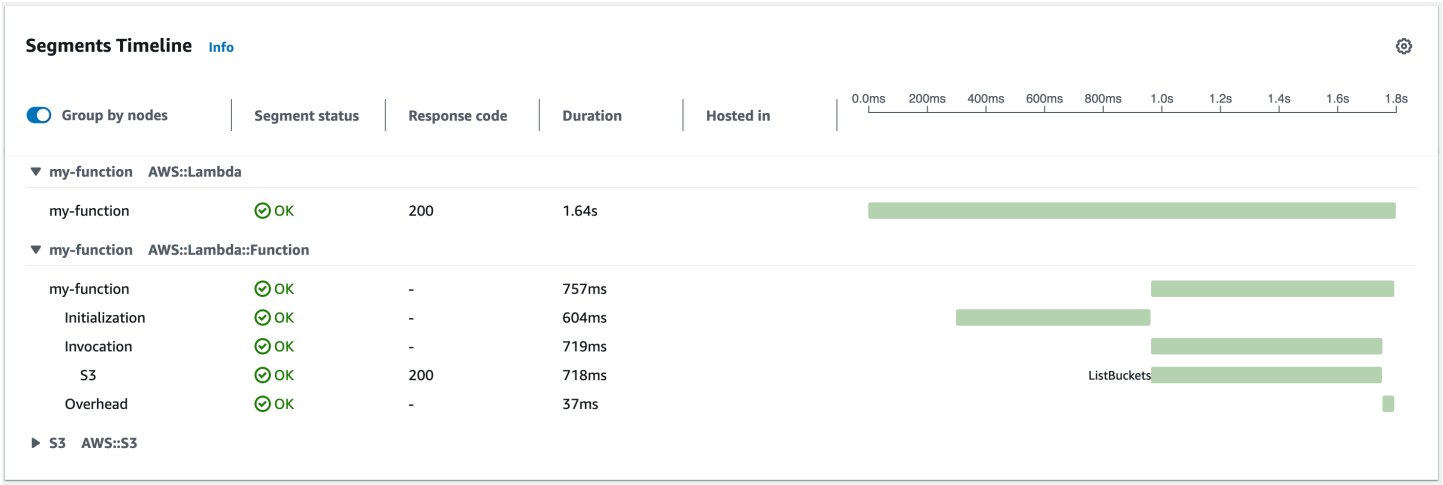
```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'
```

```
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

In X-Ray einer Ablaufverfolgung werden Informationen über eine Anfrage aufgezeichnet, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen stellen Sie möglicherweise geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollmeldungen und Trace-Segmenten fest, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter Das X-Ray SDK for Ruby im AWS X-Ray Entwicklerhandbuch.

Sections

- Aktivieren der aktiven Ablaufverfolgung mit der Lambda-API

- Aktiviert die aktive Ablaufverfolgung mit CloudFormation
- Speichern von Laufzeitabhängigkeiten in einer Ebene

Aktivieren der aktiven Ablaufverfolgung mit der Lambda-API

Verwenden Sie die folgenden API-Operationen, um die Ablaufverfolgungskonfiguration mit dem AWS SDK AWS CLI oder zu verwalten:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktiviert die aktive Ablaufverfolgung mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Speichern von Laufzeitabhängigkeiten in einer Ebene

Wenn Sie das X-Ray SDK verwenden, um SDK-Clients Ihren Funktionscode zu AWS instrumentieren, kann Ihr Bereitstellungspaket ziemlich umfangreich werden. Um zu vermeiden, dass bei jeder Aktualisierung Ihres Funktionscodes Laufzeitabhängigkeiten hochgeladen werden, packen Sie das X-Ray SDK in eine Lambda-Schicht.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion` Ressource, die X-Ray SDK for Ruby speichert.

Example template.yml — Ebene „Abhängigkeiten“

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Änderungen, um das Layer-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Beispielanwendung blank-ruby .

Lambda-Ereignisfunktionen mit Java erstellen

Sie können Java-Code in AWS Lambda Event Functions ausführen. Lambda bietet Laufzeiten für Java, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Amazon Linux-Umgebung ausgeführt, die AWS Anmeldeinformationen von einer AWS Identity and Access Management (IAM-) Rolle enthält, die Sie verwalten.

Lambda unterstützt die folgenden Java-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Java 2.5	`java25`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 21	`java21`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 17	`java17.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 11	`java11.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 8	`java8.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 17	`java17`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027
Java 11	`java11`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027
Java 8	`java8.al2`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027

AWS stellt die folgenden Bibliotheken für Java-Funktionen bereit. Diese Bibliotheken sind über Maven Central Repositoryverfügbar.

- com.amazonaws:aws-lambda-java-core (erforderlich) – definiert Handler-Methodenschnittstellen und das Kontextobjekt, das die Laufzeit an den Handler übergibt. Wenn Sie eigene Eingabetypen definieren, benötigen Sie nur diese eine Bibliothek.
- com.amazonaws:aws-lambda-java-events — Eingabetypen für Ereignisse von Diensten, die Lambda-Ereignisfunktionen aufrufen.
- com.amazonaws:aws-lambda-java-log4j2 – Eine Appender-Bibliothek für Apache Log4j 2, mit der Sie die Anfrage-ID für den aktuellen Aufruf zu Ihren Funktionsprotokollen hinzufügen können.
- AWS SDK for Java 2.0 — Das offizielle AWS SDK für die Programmiersprache Java.
- AWS Lambda Durable Execution SDK for Java — Erstellen Sie zuverlässige mehrstufige Anwendungen und KI-Workflows, die den Fortschritt überprüfen, bei Fehlern automatisch wiederherstellen und die Ausführung unterbrechen. Weitere Informationen finden Sie unter Dauerhafte Lambda-Funktionen.

Fügen Sie die Bibliotheken wie folgt der Build-Definition hinzu.

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    implementation 'software.amazon.lambda.durable:aws-durable-execution-sdk-
java:1.0.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
```

```
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
    <dependency>
      <groupId>software.amazon.lambda.durable</groupId>
      <artifactId>aws-durable-execution-sdk-java</artifactId>
      <version>1.0.1</version>
    </dependency>
  </dependencies>
```

> **Important**
>
> Verwenden Sie keine privaten Komponenten der JDK-API, wie private Felder, Methoden oder Klassen. Non-public API-Komponenten können sich bei jedem Update ändern oder entfernt werden, wodurch Ihre Anwendung nicht mehr funktioniert.

Hinweise zur benutzerdefinierten Serialisierung finden Sie unterthe section called “Benutzerdefinierte Serialisierung”.

So erstellen Sie eine Java-Ereignisfunktion

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie „Ereignisfunktion erstellen“.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Runtime: Wählen Sie Java 25.
4. Wählen Sie „Event-Funktion erstellen“.

Die Konsole erstellt eine Lambda-Funktion mit einer Handler-Klasse namens `Hello`. Da es sich bei Java um eine kompilierte Sprache handelt, können Sie den Quellcode in der Lambda-Konsole nicht anzeigen oder bearbeiten, aber Sie können seine Konfiguration ändern, ihn aufrufen und Auslöser konfigurieren.

> **Note**
>
> Um mit der Anwendungsentwicklung in Ihrer lokalen Umgebung zu beginnen, stellen Sie eine der Beispielanwendungen bereit, die im GitHub Repository dieses Handbuchs verfügbar sind.

Die `Hello`-Klasse hat eine Funktion mit dem Namen `handleRequest`, die ein Ereignisobjekt und ein Kontext-Objekt übernimmt. Dies ist die Handler-Funktion, die bei einem Aufruf der Funktion von Lambda aufgerufen wird. Die Java-Funktionslaufzeit ruft Aufrufereignisse von Lambda ab und leitet sie an den Handler weiter. In der Konfiguration der Funktion lautet der Wert für den Handler `example.Hello::handleRequest`.

Um den Funktionscode zu aktualisieren, erstellen Sie ein Bereitstellungspaket, d. h. ein ZIP-Datei-Archiv handelt, das Ihren Funktionscode enthält. Mit fortschreitender Funktionserstellung sollten Sie Ihren Funktionscode in der Versionskontrolle speichern, Bibliotheken hinzufügen und Bereitstellungen automatisieren. Beginnen Sie mit der Erstellung eines Bereitstellungspakets und der Aktualisierung Ihres Codes in der Befehlszeile.

Die Funktionslaufzeit übergibt neben dem Aufrufereignis ein Context-Objekt an den Handler. Das Context-Objekt enthält zusätzliche Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung. Weitere Informationen erhalten Sie über die Umgebungsvariablen.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- Definieren Sie den Lambda-Ereignisfunktionshandler in Java
- Stellen Sie Java Lambda Event Functions mit.zip- oder JAR-Dateiarchiven bereit
- Stellen Sie Java-Lambda-Ereignisfunktionen mit Container-Images bereit
- Arbeiten mit Ebenen für Java Lambda Event Functions
- Passen Sie die Serialisierung für Lambda Java Event Functions an
- Passen Sie das Startverhalten der Java-Laufzeit für Lambda Event Functions an
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Java-Ereignisfunktion
- Protokollieren und Überwachen von Java-Lambda-Eventfunktionen

- Instrumentierung von Java-Code in AWS Lambda
- Java-Beispielanwendungen für AWS Lambda

Definieren Sie den Lambda-Ereignisfunktionshandler in Java

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda Event Function-Handlern in Java arbeiten, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Java-Lambda-Funktion, die Informationen über einen Auftrag aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einem Bucket von Amazon Simple Storage Service (Amazon S3) platziert. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Sections

- Einrichten Ihres Java-Handler-Projekts
- Beispiel für einen Java-Lambda-Eventfunktionscode
- Gültige Klassendefinitionen für Java-Handler
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Verwendung der AWS SDK for Java v2 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für Java Lambda Event Functions

Einrichten Ihres Java-Handler-Projekts

Wenn Sie mit Lambda-Funktionen in Java arbeiten, umfasst der Prozess das Schreiben Ihres Codes, das Kompilieren und das Bereitstellen der kompilierten Artefakte in Lambda. Sie können ein Java-Lambda-Projekt auf verschiedene Arten initialisieren. Sie können beispielsweise Tools wie den Maven-Archetype für Lambda-Funktionen, den AWS SAM CLI-Befehl sam init oder sogar ein Standard-Java-Projekt-Setup in Ihrer bevorzugten IDE wie IntelliJ IDEA oder Visual Studio Code verwenden. Alternativ können Sie die erforderliche Dateistruktur manuell erstellen.

Ein typisches Java-Lambda-Funktionsprojekt folgt dieser allgemeinen Struktur:

```
/project-root
    # src
        # main
            # java
                # example
                    # OrderHandler.java (contains main handler)
                    # <other_supporting_classes>
     # build.gradle OR pom.xml
```

Sie können entweder Maven oder Gradle verwenden, um Ihr Projekt zu erstellen und Abhängigkeiten zu verwalten.

Die Haupthandler-Logik für Ihre Funktion befindet sich in einer Java-Datei im Verzeichnis `src/main/java/example`. Im Beispiel auf dieser Seite nennen wir diese Datei `OrderHandler.java`. Neben dieser Datei können Sie bei Bedarf weitere Java-Klassen hinzufügen. Achten Sie beim Bereitstellen Ihrer Funktion in Lambda darauf, die Java-Klasse anzugeben, die die Haupthandler-Methode enthält, die Lambda bei einem Aufruf ausführen soll.

Beispiel für einen Java-Lambda-Eventfunktionscode

Das folgende Beispiel für einen Java 21-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`OrderHandler.java`**Lambda-Ereignisfunktion

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;

import java.nio.charset.StandardCharsets;

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
```

```
    private static final S3Client S3_CLIENT = S3Client.builder().build();

    /**
     * Record to model the input event.
     */
    public record Order(String orderId, double amount, String item) {}

    @Override
    public String handleRequest(Order event, Context context) {
        try {
            // Access environment variables
            String bucketName = System.getenv("RECEIPT_BUCKET");
            if (bucketName == null || bucketName.isEmpty()) {
                throw new IllegalArgumentException("RECEIPT_BUCKET environment variable
is not set");
            }

            // Create the receipt content and key destination
            String receiptContent = String.format("OrderID: %s\nAmount: $%.2f\nItem:
%s",
                    event.orderId(), event.amount(), event.item());
            String key = "receipts/" + event.orderId() + ".txt";

            // Upload the receipt to S3
            uploadReceiptToS3(bucketName, key, receiptContent);

            context.getLogger().log("Successfully processed order " + event.orderId() +
                    " and stored receipt in S3 bucket " + bucketName);
            return "Success";

        } catch (Exception e) {
            context.getLogger().log("Failed to process order: " + e.getMessage());
            throw new RuntimeException(e);
        }
    }

    private void uploadReceiptToS3(String bucketName, String key, String
receiptContent) {
        try {
            PutObjectRequest putObjectRequest = PutObjectRequest.builder()
                    .bucket(bucketName)
                    .key(key)
                    .build();
```

```
            // Convert the receipt content to bytes and upload to S3
            S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to upload receipt to S3: " +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

Diese `OrderHandler.java`-Datei enthält die folgenden Abschnitte des Codes:

- `package example`: In Java kann dies beliebig sein, muss jedoch mit der Verzeichnisstruktur Ihres Projekts übereinstimmen. Hier verwenden wir `package example`, da die Verzeichnisstruktur `src/main/java/example` lautet.
- `import`-Anweisungen: Damit können Sie Java-Klassen importieren, die für Ihre Lambda-Funktion erforderlich sind.
- `public class OrderHandler ...`: Dies definiert Ihre Java-Klasse und muss eine gültige Klassendefinition sein.
- `private static final S3Client S3_CLIENT ...`: Dadurch wird ein S3-Client außerhalb einer der Methoden der Klasse initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen.
- `public record Order ...`: Definieren Sie die Form des erwarteten Eingabeereignisses in diesem benutzerdefinierten Java-Datensatz.
- `public String handleRequest(Order event, Context context)`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `private void uploadReceiptToS3(...) {}`: Dies ist eine Hilfsmethode, auf die von der `handleRequest`-Haupthandler-Methode verwiesen wird.

Beispieldateien „build.gradle“ und „pom.xml“

Die folgende `build.gradle`- oder `pom.xml`-Datei gehört zu dieser Funktion.

build.gradle

```
plugins {
    id 'java'
}
```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.3'
    implementation 'software.amazon.awssdk:s3:2.28.29'
    implementation 'org.slf4j:slf4j-nop:2.0.16'
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtimeClasspath
    }
}

java {
    sourceCompatibility = JavaVersion.VERSION_21
    targetCompatibility = JavaVersion.VERSION_21
}

build.dependsOn buildZip
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>example-java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>example-java-function</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>
```

```
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.3</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>s3</artifactId>
            <version>2.28.29</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>2.0.16</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.2</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <configuration>
                    <createDependencyReducedPom>false</createDependencyReducedPom>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*</exclude>
                                <exclude>META-INF/versions/**</exclude>
                            </excludes>
                        </filter>
                    </filters>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
```

```
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.13.0</version>
                <configuration>
                    <release>21</release>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

> **Note**
>
> Für diese Funktion sind möglicherweise zusätzliche Konfigurationseinstellungen erforderlich, damit sie ohne Timeout erfolgreich ausgeführt werden kann. Wir empfehlen, 256 MB Arbeitsspeicher und eine Zeitüberschreitung von 10 Sekunden zu konfigurieren. Der erste Aufruf kann aufgrund eines Kaltstarts zusätzliche Zeit in Anspruch nehmen. Nachfolgende Aufrufe sollten aufgrund der Wiederverwendung der Ausführungsumgebung wesentlich schneller ausgeführt werden.

Gültige Klassendefinitionen für Java-Handler

Zum Definieren Ihrer Klasse definiert die Bibliothek aws-lambda-java-core zwei Schnittstellen für Handler-Methoden. Verwenden Sie die bereitgestellten Schnittstellen, um die Handler-Konfiguration zu vereinfachen und die Methodensignatur zur Kompilierzeit zu validieren.

- com.amazonaws.services.lambda.runtime. RequestHandler
- com.amazonaws.services.lambda.runtime. RequestStreamHandler

Die `RequestHandler`-Schnittstelle ist ein generischer Typ, der zwei Parameter verwendet: den Eingabetyp und den Ausgabetyp. Beide Typen müssen Objekte sein. In diesem Beispiel implementiert unsere `OrderHandler`-Klasse `RequestHandler<OrderHandler.Order, String>`. Der Eingabetyp ist der `Order`-Datensatz, den wir innerhalb der Klasse definieren, und der Ausgabetyp ist `String`.

```
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
    ...
}
```

Wenn Sie diese Schnittstelle verwenden, deserialisiert die Java-Laufzeit das Ereignis in das Objekt mit dem Eingabetyp und serialisiert die Ausgabe in Text. Verwenden Sie diese Schnittstelle, wenn die integrierte Serialisierung mit Ihren Ein- und Ausgabetypen funktioniert.

Um Ihre eigene Serialisierung zu verwenden, können Sie die `RequestStreamHandler`-Schnittstelle implementieren. Mit dieser Schnittstelle übergibt Lamda Ihrem Handler einen Eingabestream und einen Ausgabestream. Der Handler liest Bytes aus dem Eingabestream, schreibt in den Ausgabestream und gibt „void“ zurück. Ein Beispiel dafür, wie die Java 21-Runtime verwendet wird, finden Sie unter HandlerStream.java.

Wenn Sie in Ihrer Java-Funktion nur mit grundlegenden und generischen Typen (d. h. `String` `IntegerList`,, oder`Map`) arbeiten, müssen Sie keine Schnittstelle implementieren. Wenn Ihre Funktion beispielsweise eine `Map<String, String>` Eingabe aufnimmt und eine zurückgibt`String`, könnten Ihre Klassendefinition und Handler-Signatur wie folgt aussehen:

```
public class ExampleHandler {
    public String handleRequest(Map<String, String> input, Context context) {
        ...
    }
}
```

Wenn Sie keine Schnittstelle implementieren, ist das Kontextobjekt außerdem optional. Ihre Klassendefinition und Handler-Signatur könnten beispielsweise wie folgt aussehen:

```
public class NoContextHandler {
   public String handleRequest(Map<String, String> input) {
        ...
   }
}
```

Namenskonventionen für Handler

Wenn Sie für Lambda-Funktionen in Java entweder die `RequestHandler`- oder `RequestStreamHandler`-Schnittstelle implementieren, muss Ihre Haupthandler-Methode `handleRequest` heißen. Fügen Sie außerdem das Tag `@Override` über Ihrer `handleRequest`-Methode ein. Wenn Sie Ihre Funktion in Lambda bereitstellen, geben Sie den Haupthandler in der Konfiguration Ihrer Funktion im folgenden Format an:

- *`<package>`*. *`<Class>`*— Zum Beispiel`example.OrderHandler`.

Für Lambda-Funktionen in Java, die die `RequestHandler` `RequestStreamHandler` OR-Schnittstelle nicht implementieren, können Sie einen beliebigen Namen für den Handler verwenden. Wenn Sie Ihre Funktion in Lambda bereitstellen, geben Sie den Haupthandler in der Konfiguration Ihrer Funktion im folgenden Format an:

- *`<package>`*. *`<Class>`*:: *`<handler_method_name>`* — Zum Beispiel`example.Handler::mainHandler`.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in Java 17 oder neuer arbeiten, können Sie die Form des erwarteten Eingabeereignisses als Java-Datensatz definieren. In diesem Beispiel definieren wir einen Datensatz innerhalb der `OrderHandler`-Klasse, um ein `Order`-Objekt darzustellen:

```
public record Order(String orderId, double amount, String item) {}
```

Dieser Datensatz entspricht der erwarteten Eingabeform. Nachdem Sie Ihren Datensatz definiert haben, können Sie eine Handler-Signatur schreiben, die eine JSON-Eingabe entgegennimmt, die der Datensatzdefinition entspricht. Die Java-Laufzeit deserialisiert diese JSON-Eingabe automatisch

in ein Java-Objekt. Sie können dann auf die Felder des Objekts zugreifen. Zum Beispiel holt `event.orderId` den Wert von `orderId` aus der ursprünglichen Eingabe.

> **Note**
>
> Java-Datensätze sind nur in Java 17-Laufzeiten und neueren Versionen enthalten. In allen Java-Laufzeiten können Sie eine Klasse verwenden, um Ereignisdaten darzustellen. In solchen Fällen können Sie eine Bibliothek wie jackson verwenden, um JSON-Eingaben zu deserialisieren.

Andere Typen von Eingabeereignissen

Es gibt viele mögliche Eingabeereignisse für Lambda-Funktionen in Java:

- `Integer`, `LongDouble`, und andere numerische Typen — Das Ereignis ist eine Zahl ohne zusätzliche Formatierung — zum Beispiel. `3.5` Die Java-Laufzeit wandelt den Wert in ein Objekt des angegebenen Typs um.
- `String` – Das Ereignis ist eine JSON-Zeichenfolge mit Anführungszeichen, z. B.: `"My string"`. Die Laufzeit wandelt den Wert in ein `String`-Objekt ohne Anführungszeichen um.
- `List<Integer>`, `List<String>List<Object>`, und ähnliche Typen — Das Ereignis ist ein JSON-Array. Die Laufzeitumgebung deserialisiert sie in ein Objekt des angegebenen Typs oder der angegebenen Schnittstelle.
- `InputStream` – Das Ereignis ist ein beliebiger JSON-Typ. Die Laufzeit übergibt einen Bytestream des Dokuments ohne Änderung an den Handler. Sie deserialisieren die Eingabe und schreiben Ausgabe in einen Ausgabestream.
- Bibliothekstyp — Verwenden Sie für Ereignisse, die von anderen AWS Diensten gesendet werden, die Typen in der Bibliothek aws-lambda-java-events. Wenn Ihre Lambda-Funktion beispielsweise von Amazon Simple Queue Service (SQS) aufgerufen wird, verwenden Sie das `SQSEvent`-Objekt als Eingabe.

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In diesem Beispiel ist das Kontextobjekt vom Typ `com.amazonaws.services.lambda.runtime.Context` und das zweite Argument der Haupthandlerfunktion.

```
public String handleRequest(Order event, Context context) {
    ...
}
```

Wenn Ihre Klasse entweder die RequestStreamHandlerSchnittstelle RequestHandleroder implementiert, ist das Kontextobjekt ein erforderliches Argument. Andernfalls ist das Kontextobjekt optional. Weitere Informationen zu gültigen akzeptierten Handlersignaturen finden Sie unter the section called “Gültige Klassendefinitionen für Java-Handler”.

Wenn Sie mithilfe des AWS SDK andere Dienste aufrufen, ist das Kontextobjekt in einigen wichtigen Bereichen erforderlich. Um beispielsweise Funktionsprotokolle für Amazon zu erstellen CloudWatch, können Sie die `context.getLogger()` Methode verwenden, um ein `LambdaLogger` Objekt für die Protokollierung abzurufen. In diesem Beispiel können wir den Logger verwenden, um eine Fehlermeldung zu protokollieren, falls bei der Verarbeitung aus irgendeinem Grund ein Fehler auftritt:

```
context.getLogger().log("Failed to process order: " + e.getMessage());
```

Außerhalb der Protokollierung können Sie das Kontextobjekt auch für die Funktionsüberwachung verwenden. Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Kontext”.

Verwendung der AWS SDK for Java v2 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Möglichkeit, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung des AWS SDK for Java v2.

> **Note**
>
> Das AWS SDK for Java (v1) befindet sich im Wartungsmodus und wird am 31. Dezember 2025 nicht mehr unterstützt. Wir empfehlen, dass Sie in Zukunft nur noch das AWS SDK for Java v2 verwenden.

Um SDK-Abhängigkeiten zu Ihrer Funktion hinzuzufügen, fügen Sie diese in Ihrer `build.gradle`-Datei für Gradle oder der `pom.xml`-Datei für Maven hinzu. Wir empfehlen, nur die Bibliotheken hinzuzufügen, die Sie für Ihre Funktion benötigen. Im Beispielcode zuvor haben wir die Bibliothek `software.amazon.awssdk.services.s3` verwendet. In Gradle können Sie diese Abhängigkeit

hinzufügen, indem Sie die folgende Zeile in den Abhängigkeitenabschnitt von `build.gradle` einfügen:

```
implementation 'software.amazon.awssdk:s3:2.28.29'
```

Fügen Sie in Maven die folgenden Zeilen im `<dependencies>`-Abschnitt von `pom.xml` hinzu:

```
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.28.29</version>
    </dependency>
```

> **Note**
>
> Dies ist möglicherweise nicht die neueste Version des SDK. Wählen Sie die geeignete SDK-Version für Ihre Anwendung.

Importieren Sie dann die Abhängigkeiten direkt in Ihree Java-Klasse:

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;
```

Der Beispielcode initialisiert dann einen Amazon-S3-Client wie folgt:

```
private static final S3Client S3_CLIENT = S3Client.builder().build();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS Diensten verwenden. Der Beispielcode ruft die Amazon S3 `PutObject`-API wie folgt auf:

```
PutObjectRequest putObjectRequest = PutObjectRequest.builder()
    .bucket(bucketName)
    .key(key)
    .build();
```

```
// Convert the receipt content to bytes and upload to S3
S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `System.getenv()`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
String bucketName = System.getenv("RECEIPT_BUCKET");
if (bucketName == null || bucketName.isEmpty()) {
    throw new IllegalArgumentException("RECEIPT_BUCKET environment variable is not
 set");
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code und den Klassenkonstruktor während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb der Haupt-Handler-Methode. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis ausführt, und der Client bleibt für die Wiederverwendung bei allen Aufrufen verfügbar.

Bewährte Codemethoden für Java Lambda Event Functions

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die Best Practices für Codierung anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Mit diesem Ansatz können Sie eine Funktion erstellen, die besser anhand von Einheiten getestet werden kann.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda -Ausführungsumgebung enthält eine Reihe von Bibliotheken. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, aktualisiert Lambda diese Bibliotheken regelmäßig. Diese Updates können zu subtilen Änderungen am Verhalten Ihrer Lambda-Funktion führen. Um die

Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.

- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen. Verwenden Sie beispielsweise einfachere Java Dependency Injection (IoC) Frameworks wie z. B. Dagger or Guice, als komplexere wie Spring Framework.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch wird die Zeit reduziert, die benötigt wird, bis Ihr Bereitstellungspaket vor dem Aufruf heruntergeladen und entpackt wird. Vermeiden Sie es bei in Java verfassten Funktionen, die gesamte AWS SDK-Bibliothek als Teil Ihres Bereitstellungspakets hochzuladen. Verlassen Sie sich stattdessen selektiv auf die Module, die Komponenten des benötigten SDK übernehmen (z. B. DynamoDB, Amazon S3 S3-SDK-Module und Lambda-Kernbibliotheken auf der Website). GitHub

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine

unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

- Vermeiden der Verwendung von Java-DNS-Cache. Lambda-Funktionen speichern DNS-Antworten bereits im Cache. Wenn Sie einen anderen DNS-Cache verwenden, kann es zu Verbindungs-Timeouts kommen.

  Die `java.util.logging.Logger`-Klasse kann indirekt den JVM-DNS-Cache aktivieren. Um die Standardeinstellungen zu überschreiben, setzen Sie networkaddress.cache.ttl auf 0, bevor Sie `logger` initialisieren. Beispiel:

  ```
  public class MyHandler {
    // first set TTL property
    static{
     java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
    }
   // then instantiate logger
    var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
  }
  ```

- Reduzieren Sie den Zeitaufwand, den Lambda für das Entpacken von Bereitstellungspaketen benötigt, die in Java erstellt wurden, indem Sie Ihre Abhängigkeitsdateien `.jar` in einem separaten /lib-Verzeichnis ablegen. Das ist schneller, als den gesamten Code Ihrer Funktion in ein einziges Jar mit einer großen Anzahl von Dateien zu packen. `.class` Detaillierte Anweisungen finden Sie unter Stellen Sie Java Lambda Event Functions mit.zip- oder JAR-Dateiarchiven bereit.

Stellen Sie Java Lambda Event Functions mit.zip- oder JAR-Dateiarchiven bereit

Der Code Ihrer AWS Lambda Funktion besteht aus Skripten oder kompilierten Programmen und deren Abhängigkeiten. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und ZIP-Dateiarchiven.

Auf dieser Seite wird beschrieben, wie Sie Ihr Bereitstellungspaket als ZIP-Datei oder Jar-Datei erstellen und dann das Bereitstellungspaket verwenden, um Ihren Funktionscode AWS Lambda mithilfe von AWS Command Line Interface (AWS CLI) bereitzustellen.

> **Important**
>
> In Java 25 wurde die Unterstützung für Ahead-of-Time (AOT-) Caches eingeführt. Wir empfehlen dringend, keine AOT-Caches zu verwenden, wenn Sie Ihre Funktionen als.zip- oder JAR-Dateiarchive bereitstellen, da die Caches zu unerwartetem Verhalten führen können, wenn Lambda die verwaltete Laufzeit aktualisiert. Weitere Informationen finden Sie unter. Ahead-of-Time (AOT) und CDS-Caches

Sections

- Voraussetzungen
- Tools und Bibliotheken
- Erstellen eines Bereitstellungspakets mit Gradle
- Verwenden von Ebenen für Abhängigkeiten
- Erstellen eines Bereitstellungspakets mit Maven
- Hochladen eines Bereitstellungspakets mit der Lambda-Konsole
- Hochladen eines Bereitstellungspakets mit dem AWS CLI
- Hochladen eines Bereitstellungspakets mit AWS SAM

Voraussetzungen

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Tools und Bibliotheken

AWS stellt die folgenden Bibliotheken für Java-Funktionen bereit. Diese Bibliotheken sind über Maven Central Repositoryverfügbar.

- com.amazonaws:aws-lambda-java-core (erforderlich) – definiert Handler-Methodenschnittstellen und das Kontextobjekt, das die Laufzeit an den Handler übergibt. Wenn Sie eigene Eingabetypen definieren, benötigen Sie nur diese eine Bibliothek.
- com.amazonaws:aws-lambda-java-events — Eingabetypen für Ereignisse von Diensten, die Lambda-Ereignisfunktionen aufrufen.
- com.amazonaws:aws-lambda-java-log4j2 – Eine Appender-Bibliothek für Apache Log4j 2, mit der Sie die Anfrage-ID für den aktuellen Aufruf zu Ihren Funktionsprotokollen hinzufügen können.
- AWS SDK for Java 2.0 — Das offizielle AWS SDK für die Programmiersprache Java.
- AWS Lambda Durable Execution SDK for Java — Erstellen Sie zuverlässige mehrstufige Anwendungen und KI-Workflows, die den Fortschritt überprüfen, bei Fehlern automatisch wiederherstellen und die Ausführung unterbrechen. Weitere Informationen finden Sie unter Dauerhafte Lambda-Funktionen.

Fügen Sie die Bibliotheken wie folgt der Build-Definition hinzu.

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    implementation 'software.amazon.lambda.durable:aws-durable-execution-sdk-
java:1.0.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
```

```
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
    <dependency>
      <groupId>software.amazon.lambda.durable</groupId>
      <artifactId>aws-durable-execution-sdk-java</artifactId>
      <version>1.0.1</version>
    </dependency>
  </dependencies>
```

Um ein Bereitstellungspaket zu erstellen, kompilieren Sie den Funktionscode und die Abhängigkeiten in einer einzelnen ZIP- oder Java-Archivdatei (JAR). Verwenden Sie für Gradle den `Zip`-Build-Typ. Verwenden Sie für Apache Maven das Maven Shade Plugin. Verwenden Sie zum Hochladen Ihres Bereitstellungspakets die Lambda-Konsole, die Lambda-API oder AWS Serverless Application Model ()AWS SAM.

> **Note**
>
> Um den Umfang des Bereitstellungspakets gering zu halten, verpacken Sie die Abhängigkeiten Ihrer Funktion in Ebenen. Ebenen ermöglichen Ihnen, Ihre Abhängigkeiten unabhängig zu verwalten, können von mehreren Funktionen genutzt werden und können für andere Konten zur gemeinsamen Nutzung freigegeben werden. Weitere Informationen finden Sie unter the section called “Layer”.

Erstellen eines Bereitstellungspakets mit Gradle

Um in Gradle ein Bereitstellungspaket mit dem Code Ihrer Funktion und deren Abhängigkeiten zu erstellen, verwenden Sie den Build-Typ `Zip`. Hier ist ein Beispiel aus einer vollständigen build.gradle Datei:

Example build.gradle – Entwicklungs-Aufgabe

```
task buildZip(type: Zip) {
```

```
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

Diese Build-Konfiguration erzeugt ein Bereitstellungspaket im `build/distributions`-Ordner. Innerhalb der `into('lib')`-Anweisung stellt die `jar`-Aufgabe ein JAR-Archiv mit Ihren Hauptklassen in einem Ordner mit dem Namen „`lib`“ zusammen. Die `configurations.runtimeClassPath`-Aufgabe kopiert außerdem Abhängigkeitsbibliotheken aus dem Klassenpfad des Builds in denselben Ordner mit dem Namen „`lib`“.

Example build.gradle – Abhängigkeiten

```
dependencies {
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda lädt JAR-Dateien in alphabetischer Reihenfolge (Unicode). Wenn mehrere JAR-Dateien im `lib`-Verzeichnis die gleiche Klasse enthalten, wird die erste verwendet. Sie können das folgende Shell-Skript zum Identifizieren von doppelten Klassen verwenden.

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

Verwenden von Ebenen für Abhängigkeiten

Sie können die Abhängigkeiten Ihrer Funktion in Ebenen packen, um Ihr Bereitstellungspaket klein zu halten und Abhängigkeiten unabhängig zu verwalten. Weitere Informationen finden Sie unter the section called “Layer”.

Erstellen eines Bereitstellungspakets mit Maven

Verwenden Sie das Maven Shade-Plugin, um ein Bereitstellungspaket mit Maven zu erstellen. Das Plugin erstellt eine JAR-Datei mit dem kompilierten Funktionscode und allen seinen Abhängigkeiten.

Example pom.xml – Plugin-Konfiguration

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

Verwenden Sie den `mvn package`-Befehl, um das Bereitstellungspaket zu erstellen.

```
[INFO] Scanning for projects...
[INFO] -----------------------< com.example:java-maven >-----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

Mit diesem Befehl wird eine JAR-Datei im `target`-Verzeichnis generiert.

> **Note**
>
> Wenn Sie mit einem Multi-Release-JAR (MRJAR) arbeiten, müssen Sie das MRJAR (d.h. das vom Maven Shade-Plugin erzeugte shaded JAR) in das `lib`-Verzeichnis aufnehmen und es zippen, bevor Sie Ihr Bereitstellungspaket zu Lambda hochladen. Andernfalls entpackt Lambda Ihre JAR-Datei möglicherweise nicht richtig, sodass Ihre `MANIFEST.MF` Datei ignoriert wird.

Wenn Sie die Appender-Bibliothek (`aws-lambda-java-log4j2`) verwenden, müssen Sie auch einen Transformer für das Maven Shade-Plugin konfigurieren. Die Transformer-Bibliothek kombiniert Versionen einer Cache-Datei, die sowohl in der Appender-Bibliothek als auch in Log4j angezeigt wird.

Example pom.xml – Plugin-Konfiguration mit Log4j-2-Appender

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
```

```
            </transformers>
          </configuration>
        </execution>
      </executions>
      <dependencies>
        <dependency>
          <groupId>com.github.edwgiz</groupId>
          <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
          <version>2.13.0</version>
        </dependency>
      </dependencies>
    </plugin>
```

Hochladen eines Bereitstellungspakets mit der Lambda-Konsole

Um eine neue Event-Funktion zu erstellen, müssen Sie zuerst die Funktion in der Konsole erstellen und dann Ihre .zip- oder JAR-Datei hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP- oder JAR-Datei hinzuzufügen.

Bei einer Bereitstellungspaketdatei unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-oder JAR-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

Um eine neue Event-Funktion (Konsole) zu erstellen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Create Event Function aus.
2. Wählen Sie Author from scratch aus.

3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:

    a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.

    b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.

    c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.

4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.

5. Wählen Sie „Event-Funktion erstellen“. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP- oder JAR-Archiv von Ihrem lokalen Computer (Konsole) hoch

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie die ZIP- oder JAR-Datei hochladen möchten.

2. Wählen Sie die Registerkarte Code aus.

3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.

4. Wählen Sie ZIP- oder JAR-Datei aus.

5. Laden Sie die ZIP- oder JAR-Datei wie folgt hoch:

    a. Wählen Sie Hochladen und dann Ihre ZIP- oder JAR-Datei in der Dateiauswahl aus.

    b. Klicken Sie auf Open.

    c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie eine neue .zip- oder JAR-Datei hochladen möchten.

2. Wählen Sie die Registerkarte Code aus.

3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.

4. Wählen Sie den Amazon-S3-Speicherort aus.

5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

Hochladen eines Bereitstellungspakets mit dem AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP- oder JAR-Datei zu aktualisieren. Verwenden Sie die Befehle create-function und update-function-code, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre Datei kleiner als 50 MB ist, können Sie das Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-oder JAR-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre .zip- oder JAR-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip- oder JAR-Datei mit der zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit () `--runtime` Ihrer Funktion
- Der Amazon-Ressourcenname (ARN) der Ausführungsrolle (`--role`) Ihrer Funktion
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP- oder JAR-Datei angeben. Befindet sich Ihre ZIP- oder JAR-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda create-function --function-name myFunction \
```

```
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Hochladen eines Bereitstellungspakets mit AWS SAM

Sie können AWS SAM es verwenden, um die Bereitstellung Ihres Funktionscodes, Ihrer Konfiguration und Ihrer Abhängigkeiten zu automatisieren. AWS SAM ist eine Erweiterung von CloudFormation , die eine vereinfachte Syntax für die Definition serverloser Anwendungen bietet. Die folgende Beispielvorlage definiert eine Funktion mit einem Bereitstellungspaket in dem `build/distributions`-Verzeichnis, das Gradle verwendet.

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
```

```
      Handler: example.Handler
      Runtime: java25
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

Um die Funktion zu erstellen, verwenden Sie die Befehle „`package`“ und „`deploy`“. Bei diesen Befehlen handelt es sich um Anpassungen an der AWS CLI. Sie umschließen andere Befehle, um das Bereitstellungspaket zu Amazon S3 hochzuladen, die Vorlage mit dem Objekt-URI neu zu schreiben und den Code der Funktion zu aktualisieren.

Im folgenden Beispielskript wird ein Gradle-Build ausgeführt und das von ihm erstellte Bereitstellungspaket wird hochgeladen. Es erstellt einen CloudFormation Stack, wenn Sie es zum ersten Mal ausführen. Wenn der Stack bereits vorhanden ist, wird der vom Skript aktualisiert.

Example deploy.sh

```
#!/bin/bash
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

Ein vollständiges funktionierendes Beispiel finden Sie in den folgenden Beispielanwendungen.

Lambda-Beispielanwendungen in Java

- example-java — Eine Java-Event-Funktion, die demonstriert, wie Sie Lambda zur Bearbeitung von Bestellungen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das SDK verwenden und wie Sie die AWS Ausgabe protokollieren.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.

- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Stellen Sie Java-Lambda-Ereignisfunktionen mit Container-Images bereit

Es gibt drei Möglichkeiten, ein Container-Image für eine Java-Lambda-Ereignisfunktion zu erstellen:

- Verwenden eines AWS Basis-Image für Java

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Mit einem AWS OS-only Basisbild

  AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Java in das Image aufnehmen.

- Verwenden Sie ein Image, das nicht zur AWS Basisversion gehört

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Java in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für Java
- Verwenden eines AWS Basis-Image für Java

- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für Java

AWS stellt die folgenden Basis-Images für Java bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
25	Java 2.5	Amazon Linux 2023	Dockerfile für Java 25 auf GitHub	30. Juni 2029
21	Java 21	Amazon Linux 2023	Dockerfile für Java 21 auf GitHub	30. Juni 2029
17. April 2023	Java 17	Amazon Linux 2023	Dockerfile für Java 17 auf GitHub	30. Juni 2029
11. April 2023	Java 11	Amazon Linux 2023	Dockerfile für Java 11 auf GitHub	30. Juni 2029
8. April 2023	Java 8	Amazon Linux 2023	Dockerfile für Java 8 auf GitHub	30. Juni 2029
17	Java 17	Amazon Linux 2	Dockerfile für Java 17 auf GitHub	30. Juni 2027
11	Java 11	Amazon Linux 2	Dockerfile für Java 11 auf GitHub	30. Juni 2027
8.al2	Java 8	Amazon Linux 2	Dockerfile für Java 8 auf GitHub	30. Juni 2027

Amazon ECR-Repository: gallery.ecr. aws/lambda/java

Die Basis-Images für Java 21 und höher basieren auf dem Amazon Linux 2023 Minimal Container Image. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie `glibc`.

AL2023-based Bilder verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, was der Standard-Paketmanager in Amazon Linux 2 ist. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2023-based Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2023 und Amazon Linux 2 finden Sie unter Introducing the Amazon Linux 2023 runtime for AWS Lambda im AWS -Compute-Blog.

> **Note**
>
> Um AL2023-based Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Verwenden eines AWS Basis-Image für Java

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java (zum Beispiel Amazon Corretto)
- Apache Maven oder Gradle
- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen eines Images aus einem Base Image

Maven

1. Führen Sie den folgenden Befehl aus, um ein Maven-Projekt mit dem Archetyp für Lambda zu erstellen. Die folgenden Parameter sind erforderlich:

- service — Der AWS-Service Client, der in der Lambda-Funktion verwendet werden soll. Eine Liste der verfügbaren Quellen finden Sie unter aws-sdk-java - on. v2/services GitHub
- region — Der AWS-Region Ort, an dem Sie die Lambda-Funktion erstellen möchten.
- groupId – Der vollständige Paket-Namespace Ihrer Anwendung.
- artifactId – Ihr Projektname. Dies wird der Name des Verzeichnisses für Ihr Projekt.

Führen Sie unter Linux und macOS folgenden Befehl aus:

```
mvn -B archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
   -DgroupId=com.example.myapp \
   -DartifactId=myapp
```

Führen Sie in PowerShell diesen Befehl aus:

```
mvn -B archetype:generate `
   "-DarchetypeGroupId=software.amazon.awssdk" `
   "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
 `
   "-DgroupId=com.example.myapp" `
   "-DartifactId=myapp"
```

Der Maven-Archetyp für Lambda ist für die Kompilierung mit Java SE 8 vorkonfiguriert und enthält eine Abhängigkeit von der AWS SDK für Java. Wenn Sie Ihr Projekt mit einem anderen Archetyp oder mit einer anderen Methode erstellen, müssen Sie den Java-Compiler für Maven konfigurieren und das SDK als Abhängigkeit deklarieren.

2. Öffnen Sie das *myapp*`/src/main/java/com/example/`*myapp*-Verzeichnis, und suchen Sie die `App.java`-Datei. Dies ist der Code für die Lambda-Funktion. Sie können den bereitgestellten Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.
3. Navigieren Sie zurück zum Stammverzeichnis des Projekts und erstellen Sie ein neues Dockerfile mit der folgenden Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Maven layout
COPY target/classes ${LAMBDA_TASK_ROOT}
COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.myapp.App::handleRequest" ]
```

4. Kompilieren Sie das Projekt und erfassen Sie die Laufzeitabhängigkeiten.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

Gradle

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir example
cd example
```

2. Führen Sie den folgenden Befehl aus, um Gradle ein neues Java-Anwendungsprojekt im Verzeichnis `example` der Umgebung generieren zu lassen. Wählen Sie für Select Build Script DSL die Option 2: Groovy aus.

```
gradle init --type java-application
```

3. Öffnen Sie das */example*/app/src/main/java/*example*-Verzeichnis, und suchen Sie die `App.java`-Datei. Dies ist der Code für die Lambda-Funktion. Sie können den folgenden Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

   Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. Öffnen Sie die `build.gradle` Datei. Wenn Sie den Beispiel-Funktionscode aus dem vorherigen Schritt verwenden, ersetzen Sie den Inhalt von `build.gradle` durch den folgenden. Wenn Sie Ihren eigenen Funktionscode verwenden, ändern Sie Ihre `build.gradle`-Datei nach Bedarf.

   Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
```

```
repositories {
  mavenCentral()
}
dependencies {
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
jar {
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. Der `gradle init`-Befehl aus Schritt 2 generierte auch einen Dummy-Testfall im `app/test`-Verzeichnis. Überspringen Sie in diesem Tutorial das Ausführen von Tests, indem Sie das `/test`-Verzeichnis löschen.
6. Erstellen Sie das Projekt.

```
gradle build
```

7. Erstellen Sie im Stammverzeichnis (`/example`) des Projekts eine Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in eine Lambda-defined Umgebungsvariable zu `{LAMBDA_TASK_ROOT}` kopieren.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}
```

```
# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

Linux/macOS

Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

   Wenn erfolgreich, sehen Sie eine Antwort wie diese:

   ```
   {
       "repository": {
           "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
   world",
           "registryId": "111122223333",
           "repositoryName": "hello-world",
           "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world",
           "createdAt": "2023-03-09T10:39:01+00:00",
           "imageTagMutability": "MUTABLE",
           "imageScanningConfiguration": {
               "scanOnPush": true
   ```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-only Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Runtime-Interface-Client in Ihr Image aufnehmen. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Laufzeitschnittstellen-Client für Java in Ihrem Dockerfile oder als Abhängigkeit in Ihrem Projekt. Um beispielsweise den Laufzeitschnittstellen-Client mit dem Maven-Paketmanager zu installieren, fügen Sie Ihrer `pom.xml`-Datei Folgendes hinzu:

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

Einzelheiten zum Paket finden Sie unter AWS Lambda Java Runtime Interface Client im Maven-Central-Repository. Sie können den Quellcode des Runtime-Interface-Clients auch im GitHub Repository der AWS Lambda Java Support Libraries überprüfen.

Das folgende Beispiel zeigt, wie Sie mithilfe eines Amazon-Corretto-Images ein Container-Image für Java erstellen. Amazon Corretto ist eine kostenlose, plattformübergreifende und produktionsbereite Distribution des Open Java Development Kit (OpenJDK). Das Maven-Projekt beinhaltet den Laufzeitschnittstellen-Client als Abhängigkeit.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java (zum Beispiel Amazon Corretto)
- Apache Maven
- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen eines Images aus einem alternativen Basis-Image

1. Erstellen Sie ein Maven-Projekt. Die folgenden Parameter sind erforderlich:
    - groupId – Der vollständige Paket-Namespace Ihrer Anwendung.

- artifactId – Ihr Projektname. Dies wird der Name des Verzeichnisses für Ihr Projekt.

Linux/macOS

```
mvn -B archetype:generate \
   -DarchetypeArtifactId=maven-archetype-quickstart \
   -DgroupId=example \
   -DartifactId=myapp \
   -DinteractiveMode=false
```

PowerShell

```
mvn -B archetype:generate `
   -DarchetypeArtifactId=maven-archetype-quickstart `
   -DgroupId=example `
   -DartifactId=myapp `
   -DinteractiveMode=false
```

2. Öffnen Sie das Projektverzeichnis.

```
cd myapp
```

3. Öffnen Sie die Datei `pom.xml` und ersetzen Sie den Inhalt durch Folgendes. Diese Datei enthält aws-lambda-java-runtime-interface-client als Abhängigkeit. Alternativ können Sie den Laufzeitschnittstellen-Client im Dockerfile installieren. Der einfachste Ansatz besteht jedoch darin, die Bibliothek als Abhängigkeit einbinden.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>example</groupId>
  <artifactId>hello-lambda</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-lambda</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
```

```
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. Öffnen Sie das *myapp*`/src/main/java/com/example/`*myapp*-Verzeichnis, und suchen Sie die `App.java`-Datei. Dies ist der Code für die Lambda-Funktion. Ersetzen Sie den Code durch Folgendes:

   Example Funktions-Handler

   ```
   package example;

   public class App {
       public static String sayHello() {
           return "Hello world!";
       }
   }
   ```

5. Der `mvn -B archetype:generate`-Befehl aus Schritt 1 generierte auch einen Dummy-Testfall im `src/test`-Verzeichnis. Überspringen Sie in diesem Tutorial das Ausführen von Tests, indem Sie das gesamte generierte Verzeichnis `/test` löschen.
6. Navigieren Sie zurück zum Stammverzeichnis des Projekts und erstellen Sie dann ein neues Dockerfile. Das folgende Beispiel-Dockerfile verwendet ein Amazon-Corretto-Image. Amazon Corretto ist eine kostenlose, plattformübergreifende und produktionsbereite Distribution des OpenJDK.

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function
```

```
COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello" ]
```

7. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

- Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
- Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

   > **Note**
   >
   > Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Java Lambda Event Functions

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Fügen Sie die Ebene zu Ihrer Funktion hinzu

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Stellen Sie sicher, dass die Java-Version, auf die sich Maven oder Gradle bezieht, mit der Java-Version der Funktion übereinstimmt, die Sie bereitstellen möchten. Bei einer Java 25-Funktion sollte der `mvn -v` Befehl beispielsweise Java 25 in der Ausgabe auflisten.
- Ihre Abhängigkeiten müssen im Verzeichnis `java/lib` im Stammverzeichnis der ZIP-Datei gespeichert werden. Weitere Informationen finden Sie unter Ebenenpfade für jede Lambda-Laufzeit.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder Java-Bibliotheken von Drittanbietern oder Ihre eigenen Java-Module und -Pakete enthalten. Im Folgenden wird Maven verwendet. Sie können auch Gradle verwenden, um den Inhalt Ihrer Ebenen zu verpacken.

Erstellen einer Ebene mit Maven-Abhängigkeiten

1. Erstellen Sie ein Apache-Maven-Projekt mit einer `pom.xml`-Datei, die Ihre Abhängigkeiten definiert.

   Das folgende Beispiel enthält Jackson Databind für die JSON-Verarbeitung. Der Abschnitt `<build>` verwendet das maven-dependency-plugin, um für jede Abhängigkeit separate JAR-Dateien zu erstellen, anstatt sie in einem einzigen Uber-JAR zu bündeln. Wenn Sie ein Uber-Jar erstellen möchten, verwenden Sie das maven-shade-plugin.

   Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.13.0</version>
            <configuration>
                <source>21</source>
                <target>21</target>
                <release>21</release>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.6.1</version>
            <executions>
                <execution>
                    <id>copy-dependencies</id>
                    <phase>package</phase>
                    <goals>
                        <goal>copy-dependencies</goal>
```

```
                    </goals>
                    <configuration>
                        <outputDirectory>${project.build.directory}/lib</
outputDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

2. Erstellen Sie das Projekt. Dieser Befehl erstellt alle JAR-Abhängigkeitsdateien im `target/lib/`-Verzeichnis.

```
mvn clean package
```

3. Erstellen Sie die erforderliche Verzeichnisstruktur für Ihre Ebene:

```
mkdir -p java/lib
```

4. Kopieren Sie die JAR-Abhängigkeitsdateien in das `java/lib`-Verzeichnis:

```
cp target/lib/*.jar java/lib/
```

5. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip java/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\java -DestinationPath .\layer.zip
   ```

   Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
java/
### lib/
    ### jackson-databind-2.17.0.jar
    ### jackson-core-2.17.0.jar
```

```
### jackson-annotations-2.17.0.jar
```

> **Note**
>
> Stellen Sie sicher, dass Ihre ZIP-Datei das `java`-Verzeichnis auf der Stammebene enthält und sich `lib` darin befindet. Diese Struktur stellt sicher, dass Lambda Ihre Bibliotheken finden und importieren kann. Jede Abhängigkeit wird als separate JAR-Datei gespeichert und nicht in einem Uber-JAR gebündelt.

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den AWS CLI Befehl publish-layer-version aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes java25
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.
4. (Optional) Wählen Sie für Kompatible Laufzeiten die Java-Laufzeit aus, die der Java-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Um die Ebene an Ihre Funktion anzuhängen, führen Sie den Befehl update-function-configuration aus. AWS CLI Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

   > **Note**
   >
   > Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Passen Sie die Serialisierung für Lambda Java Event Functions an

Die verwalteten Lambda-Java-Laufzeiten unterstützen die benutzerdefinierte Serialisierung für JSON-Ereignisse. Eine benutzerdefinierte Serialisierung kann Ihren Code vereinfachen und möglicherweise die Leistung verbessern.

Themen

- Wann sollte die benutzerdefinierte Serialisierung verwendet werden
- Benutzerdefinierte Serialisierung implementieren
- Testen der benutzerdefinierten Serialisierung

Wann sollte die benutzerdefinierte Serialisierung verwendet werden

Wenn Ihre Lambda-Funktion aufgerufen wird, müssen die Eingabeereignisdaten in ein Java-Objekt deserialisiert werden und die Ausgabe Ihrer Funktion muss wieder in ein Format serialisiert werden, das als Antwort der Funktion zurückgegeben werden kann. Die von Lambda Java verwalteten Runtimes bieten standardmäßige Serialisierungs- und Deserialisierungsfunktionen, die sich gut für die Verarbeitung von Ereignisnutzlasten verschiedener AWS Dienste wie Amazon API Gateway und Amazon Simple Queue Service (Amazon SQS) eignen. Um mit diesen Serviceintegrationsereignissen in Ihrer Funktion zu arbeiten, fügen Sie Ihrem Projekt die Abhängigkeit aws-java-lambda-events hinzu. Diese AWS Bibliothek enthält Java-Objekte, die diese Service-Integrationsereignisse darstellen.

Sie können auch Ihre eigenen Objekte verwenden, um den Event-JSON darzustellen, den Sie an Ihre Lambda-Funktion übergeben. Die verwaltete Laufzeit versucht, den JSON mit seinem Standardverhalten in eine neue Instance Ihres Objekts zu serialisieren. Wenn der Standard-Serializer nicht das gewünschte Verhalten für Ihren Anwendungsfall aufweist, verwenden Sie die benutzerdefinierte Serialisierung.

Angenommen, eine `Vehicle`-Klasse mit dem Namen besitzt die folgende Verzeichnisstruktur:

```
public class Vehicle {
    private String vehicleType;
    private long vehicleId;
}
```

Die Nutzdaten des JSON-Ereignisses sieht jedoch wie folgt aus:

```
{
```

```
    "vehicle-type": "car",
    "vehicleID": 123
}
```

In diesem Szenario erwartet die Standard-Serialisierung in der verwalteten Laufzeitumgebung, dass die JSON-Eigenschaftsnamen mit den Eigenschaftsnamen der Java-Klassen (`vehicleType`, `vehicleId`) in Kamelform übereinstimmen. Da die Eigenschaftsnamen im JSON-Ereignis nicht in Großbuchstaben (`vehicle-type`, `vehicleID`) geschrieben sind, müssen Sie eine benutzerdefinierte Serialisierung verwenden.

Benutzerdefinierte Serialisierung implementieren

Verwenden Sie anstelle der Standard-Serialisierungslogik der verwalteten Runtime ein Service Provider-Interface, um einen Serializer Ihrer Wahl zu laden. Mithilfe der `RequestHandler`-Standardschnittstelle können Sie Ihre JSON-Event-Nutzdaten direkt in Java-Objekte serialisieren.

So verwenden Sie die benutzerdefinierte Serialisierung in Ihrer Java Lambda Event Function

1. Fügen Sie die aws-lambda-java-core-Bibliothek als Abhängigkeit hinzu. Diese Bibliothek enthält die CustomPojoSerializerSchnittstelle zusammen mit anderen Schnittstellendefinitionen für die Arbeit mit Java in Lambda.
2. Erstellen Sie eine Datei namens `com.amazonaws.services.lambda.runtime.CustomPojoSerializer` im Verzeichnis `src/main/resources/META-INF/services/` Ihres Projekts.
3. Geben Sie in dieser Datei den vollqualifizierten Namen Ihrer benutzerdefinierten Serializer-Implementierung an, die die `CustomPojoSerializer`-Schnittstelle implementieren muss. Beispiel:

   ```
   com.mycompany.vehicles.CustomLambdaSerialzer
   ```
4. Implementieren Sie die `CustomPojoSerializer`-Schnittstelle, um Ihre benutzerdefinierte Serialisierungslogik bereitzustellen.
5. Verwenden Sie die `RequestHandler`-Standardschnittstelle in Ihrer Lambda-Funktion. Die verwaltete Laufzeit verwendet Ihren benutzerdefinierten Serializer.

Weitere Beispiele für die Implementierung benutzerdefinierter Serialisierung mithilfe gängiger Bibliotheken wie FastJSON, Gson, Moshi und jackson-jr finden Sie im Beispiel für die benutzerdefinierte Serialisierung im Repository. AWS GitHub

Testen der benutzerdefinierten Serialisierung

Testen Sie Ihre Funktion, um sicherzustellen, dass Ihre Serialisierungs- und Deserialisierungslogik wie erwartet funktioniert. Sie können die AWS Serverless Application Model Befehlszeilenschnittstelle (AWS SAMCLI) verwenden, um den Aufruf Ihrer Lambda-Payload zu emulieren. Dies kann Ihnen helfen, Ihre Funktion schnell zu testen und zu iterieren, wenn Sie einen benutzerdefinierten Serializer einführen.

1. Erstellen Sie eine Datei mit der JSON-Ereignisnutzlast, mit der Sie Ihre Funktion aufrufen möchten, und rufen Sie dann die auf. AWS SAMCLI
2. Führen Sie denselben lokalen Invoke-Befehl aus, um Ihre Funktion lokal aufzurufen. Beispiel:

```
sam local invoke -e src/test/resources/event.json
```

Weitere Informationen finden Sie unter Lambda-Funktionen lokal aufrufen mit. AWS SAM

Passen Sie das Startverhalten der Java-Laufzeit für Lambda Event Functions an

Auf dieser Seite werden die spezifischen Einstellungen für Java-Ereignisfunktionen in AWS Lambda beschrieben. Sie können diese Einstellungen verwenden, um das Startup-Verhalten der Java-Laufzeit anzupassen. Dies kann die allgemeine Funktionslatenz reduzieren und die Gesamtleistung der Funktionen verbessern, ohne dass Code geändert werden muss.

Sections

- Grundlegendes zur Umgebungsvariablen JAVA_TOOL_OPTIONS
- Log4j-Patch für Log4Shell
- Ahead-of-Time (AOT) und CDS-Caches

Grundlegendes zur Umgebungsvariablen **`JAVA_TOOL_OPTIONS`**

In Java unterstützt Lambda die Umgebungsvariable `JAVA_TOOL_OPTIONS`, um zusätzliche Befehlszeilenvariablen in Lambda festzulegen. Sie können diese Umgebungsvariable auf verschiedene Arten verwenden, z. B. um die Einstellungen für die gestufte Kompilierung anzupassen. Im nächsten Beispiel wird gezeigt, wie die Umgebungsvariable `JAVA_TOOL_OPTIONS` für diesen Anwendungsfall verwendet wird.

Beispiel: Einstellungen für die gestufte Kompilierung anpassen

Die gestufte Kompilierung ist ein Feature der virtuellen Java-Maschine (JVM). Sie können spezielle Einstellungen für die gestufte Kompilierung verwenden, um die Just-in-Time (JIT)-Compiler der JVM optimal zu nutzen. In der Regel ist der C1-Compiler für eine schnelle Startzeit optimiert. Der C2-Compiler ist für die beste Gesamtleistung optimiert, benötigt aber auch mehr Speicher und es dauert länger, bis er erreicht ist.

Die gestufte Kompilierung umfasst 5 verschiedene Stufen. Auf Stufe 0 interpretiert die JVM Java-Bytecode. Auf Stufe 4 verwendet die JVM den C2-Compiler, um die beim Startup der Anwendung gesammelten Profildaten zu analysieren. Im Laufe der Zeit überwacht sie die Code-Nutzung, um die besten Optimierungen zu identifizieren.

Das Anpassen der mehrstufigen Kompilierungsebene kann Ihnen helfen, die Leistung Ihrer Java-Funktion zu optimieren. Bei kleinen Funktionen, die schnell ausgeführt werden, kann das Setzen der mehrstufigen Kompilierung auf Stufe 1 dazu beitragen, die Kaltstartleistung zu verbessern, indem die JVM den C1-Compiler verwendet. Diese Einstellung erzeugt schnell optimierten systemeigenen Code, generiert jedoch keine Profilerstellungsdaten und verwendet niemals den C2-

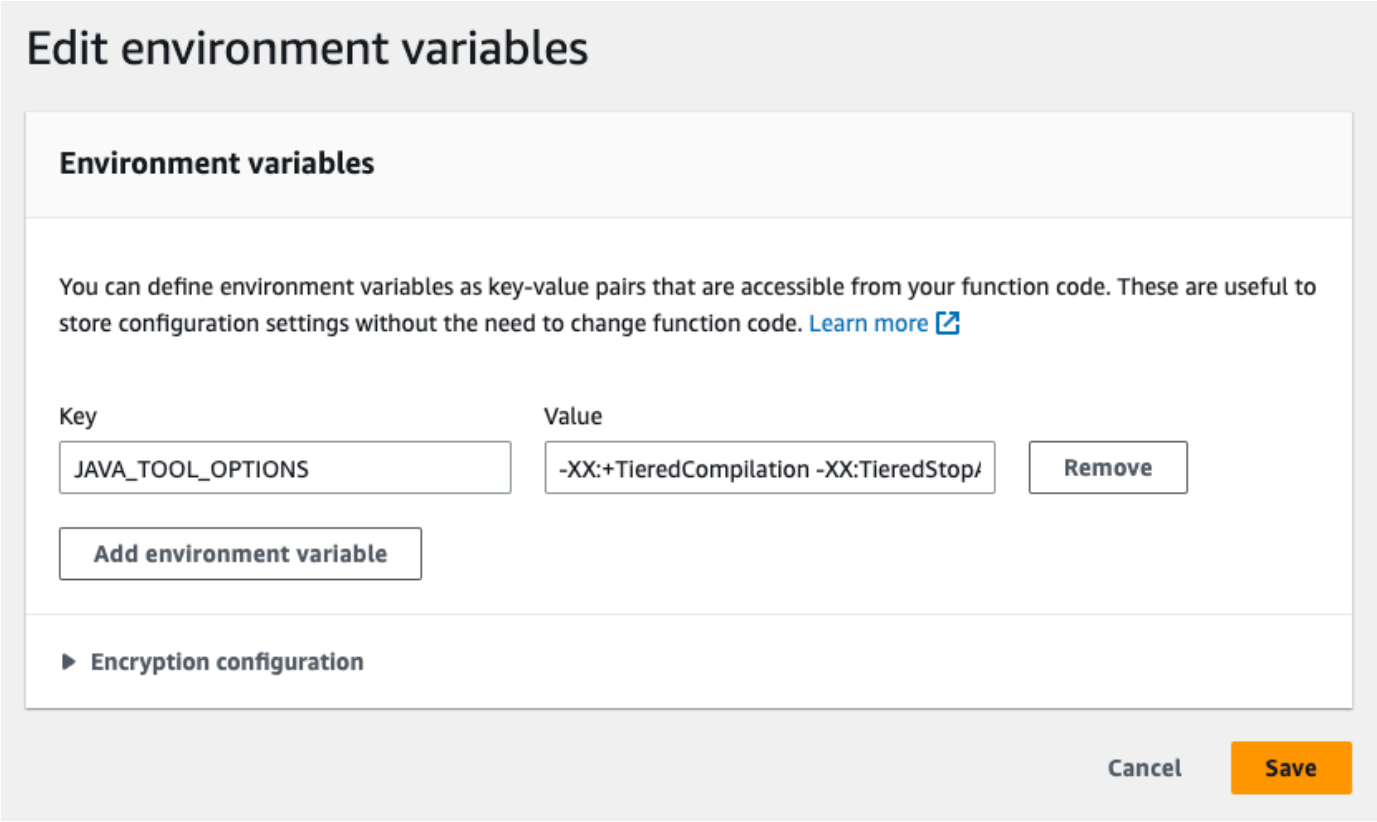
Compiler. Bei größeren, rechenintensiven Funktionen maximiert die Einstellung der mehrstufigen Kompilierung auf Stufe 4 die Gesamtleistung auf Kosten von zusätzlichem Speicherverbrauch und zusätzlicher Optimierungsarbeit bei den ersten Aufrufen nach der Bereitstellung jeder Lambda-Ausführungsumgebung.

Für Java 11-Laufzeiten und darunter verwendet Lambda die standardmäßigen JVM-Einstellungen für die mehrstufige Kompilierung. Für Java 17 und Java 21 konfiguriert Lambda die JVM so, dass die mehrstufige Kompilierung standardmäßig auf Stufe 1 beendet wird. Ab Java 25 stoppt Lambda die mehrstufige Kompilierung immer noch standardmäßig auf Stufe 1, außer wenn Parallelität SnapStart oder Provisioned verwendet wird. In diesem Fall werden die standardmäßigen JVM-Einstellungen verwendet.

Dadurch wird die Leistung bei Parallelität SnapStart und Bereitstellung verbessert, ohne dass es zu einer Kaltstartstrafe kommt, da die gestaffelte Kompilierung in diesen Fällen außerhalb des Aufrufpfads erfolgt. Um diesen Vorteil zu maximieren, können Sie das Priming — das Ausführen von Codepfaden während der Funktionsinitialisierung — verwenden, um JIT auszulösen, bevor der SnapStart Snapshot erstellt wird, oder wenn die Ausführungsumgebungen von Provisioned Concurrency vorab bereitgestellt werden. Weitere Informationen finden Sie im Blogbeitrag Optimierung der Kaltstartleistung von AWS Lambda mithilfe fortschrittlicher Priming-Strategien mit. SnapStart

Anpassen der Einstellungen für die gestufte Kompilierung (Konsole)

1. Öffnen Sie die Seite Event Functions in der Lambda-Konsole.
2. Wählen Sie eine Java-Funktion, für die Sie die gestufte Kompilierung anpassen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann im linken Menü die Option Umgebungsvariablen aus.
4. Wählen Sie Bearbeiten aus.
5. Wählen Sie Umgebungsvariablen hinzufügen aus.
6. Geben Sie `JAVA_TOOL_OPTIONS` als Schlüssel ein. Geben Sie `-XX:+TieredCompilation -XX:TieredStopAtLevel=1` als Wert ein.



7. Wählen Sie Speichern.

> **Note**
>
> Sie können Lambda auch verwenden, SnapStart um Probleme mit dem Kaltstart zu beheben. SnapStartverwendet zwischengespeicherte Snapshots Ihrer Ausführungsumgebung, um die Startleistung erheblich zu verbessern. Weitere Informationen zu SnapStart Funktionen, Einschränkungen und unterstützten Regionen finden Sie unter. Verbesserung der Startleistung mit Lambda SnapStart

Beispiel: Anpassen des GC-Verhaltens mit JAVA_TOOL_OPTIONS

Java-11-Laufzeiten verwenden den Serial Garbage Collector (GC) für Garbage Collection. Standardmäßig verwenden Java-17-Laufzeiten auch den Serial GC. Mit Java 17 können Sie jedoch auch die Umgebungsvariable `JAVA_TOOL_OPTIONS` verwenden, um den Standard-GC zu ändern. Sie können zwischen dem Parallel GC und dem Shenandoah GC wählen.

Wenn Ihre Workload beispielsweise mehr Arbeitsspeicher und mehrere CPUs benötigt, sollten Sie erwägen, den Parallel GC zu verwenden, um eine bessere Leistung zu erzielen. Hängen Sie dazu Folgendes an den Wert Ihrer Umgebungsvariablen `JAVA_TOOL_OPTIONS` an:

```
-XX:+UseParallelGC
```

Wenn Ihr Workload viele kurzlebige Objekte enthält, können Sie von einem geringeren Speicherverbrauch profitieren, indem Sie den in Java 25 eingeführten Generationsmodus des Shenandoah-Garbage-Collectors aktivieren. Hängen Sie dazu Folgendes an den Wert Ihrer Umgebungsvariablen an: `JAVA_TOOL_OPTIONS`

```
-XX:+UseShenandoahGC -XX:ShenandoahGCMode=generational
```

Log4j-Patch für Log4Shell

Die Lambda-Laufzeiten für Java 8, 11, 17 und 21 enthalten einen Patch zur Minderung der Log4Shell-Sicherheitslücke (CVE-2021-44228) in Log4j, einem beliebten Java-Logging-Framework. Dieser Patch verursacht einen Leistungsaufwand beim Kaltstart. Wenn Sie eine gepatchte Version von Log4j (Version 2.17.0 oder höher) verwenden, können Sie diesen Patch deaktivieren, um die Kaltstartleistung zu verbessern. Um den Patch zu deaktivieren, setzen Sie die Umgebungsvariable auf. `AWS_LAMBDA_DISABLE_CVE_2021_44228_PROTECTION` `true`

Ab Java 25 enthalten Lambda-Laufzeiten den Log4Shell-Patch nicht mehr. Sie müssen überprüfen, ob Sie Log4j Version 2.17.0 oder höher verwenden.

Ahead-of-Time (AOT) und CDS-Caches

Ab Java 25 umfasst die Lambda-Laufzeit einen Ahead-of-Time (AOT-) Cache für den Java Runtime Interface Client (RIC), eine Laufzeitkomponente, die aktiv nach Ereignissen von der Lambda Runtime API abfragt. Dies verbessert die Kaltstartleistung.

AOT-Caches sind spezifisch für einen JVM-Build. Wenn Lambda die verwaltete Laufzeit aktualisiert, aktualisiert es auch den AOT-Cache für das RIC. Wenn Sie jedoch Ihre eigenen AOT-Caches bereitstellen, können diese ungültig werden oder nach einem Runtime-Update zu unerwartetem Verhalten führen. Wir empfehlen daher dringend, keine AOT-Caches zu verwenden, wenn Sie verwaltete Laufzeiten verwenden. Um AOT-Caches zu verwenden, sollten Sie Ihre Funktionen mithilfe von Container-Images bereitstellen.

AOT-Caches können nicht mit Class Data Sharing (CDS) -Caches verwendet werden. Wenn Sie CDS-Caches in Ihrer Lambda-Funktion bereitstellen, deaktiviert Lambda den AOT-Cache.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Java-Ereignisfunktion

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Methoden

- `getRemainingTimeInMillis()` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.
- `getFunctionName()` – Gibt den Namen der Lambda-Funktion zurück.
- `getFunctionVersion()` – Gibt die Version der Funktion zurück.
- `getInvokedFunctionArn()` – Gibt den Amazon-Ressourcennamen (ARN) zurück, der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `getMemoryLimitInMB()` – Gibt die Menge an Arbeitsspeicher zurück, die der Funktion zugewiesen ist.
- `getAwsRequestId()` – Gibt den Bezeichner der Aufrufanforderung zurück.
- `getLogGroupName()` – Gibt die Protokollgruppe für die Funktion zurück.
- `getLogStreamName()` – Gibt den Protokollstrom für die Funktionsinstance zurück.
- `getIdentity()` – Gibt Informationen zur Amazon-Cognito-Identität zurück, die die Anforderung autorisiert hat.
- `getClientContext()` – (mobile Apps) Gibt Clientkontext zurück, der Lambda von der Clientanwendung bereitgestellt wird.
- `getLogger()` – Gibt das Logger-Objekt für die Funktion zurück.

Das folgende Beispiel zeigt eine Funktion, die das Kontextobjekt verwendet, um auf den Lambda-Logger zuzugreifen.

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

Die Funktion protokolliert den Klassentyp des eingehenden Ereignisses, bevor sie `null` zurückgibt.

Example Protokollausgabe

```
EVENT TYPE: class java.util.LinkedHashMap
```

Die Schnittstelle für das Kontextobjekt ist in der aws-lambda-java-core-Bibliothek verfügbar. Sie können diese Schnittstelle implementieren, um eine Kontextklasse zum Testen zu erstellen. Das folgende Beispiel zeigt eine Kontextklasse, die Dummy-Werte für die meisten Eigenschaften und einen funktionierenden Testlogger zurückgibt.

Example src/test/java/example/TestContext.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
```

```
  }
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
  public String getInvokedFunctionArn(){
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

Weitere Informationen zu Protokollierung finden Sie unter Protokollieren und Überwachen von Java-Lambda-Eventfunktionen.

Kontext in Beispielanwendungen

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung des Kontextobjekts demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine AWS Serverless Application Model (AWS SAM) -Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java — Eine Java-Event-Funktion, die demonstriert, wie Sie Lambda zur Bearbeitung von Bestellungen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das SDK verwenden und wie Sie die AWS Ausgabe protokollieren.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Protokollieren und Überwachen von Java-Lambda-Eventfunktionen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterLambda Event Function-Logs an CloudWatch Logs senden.

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie Methoden von java.lang.System oder jedes Protokollierungsmodul verwenden, das nach stdout oder stderr schreibt.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java
- Implementierung der erweiterten Protokollierung mit Log4j2 und SLF4J
- Andere Tools und Bibliotheken für die Protokollierung verwenden
- Verwendung von Powertools für AWS Lambda (Java) und AWS SAM für strukturiertes Logging
- Anzeigen von Protokollen in der Lambda-Konsole
- Logs in der CloudWatch Konsole anzeigen
- Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)
- Löschen von Protokollen
- Beispielprotokolliercode

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie Methoden für java.lang.System verwenden oder ein beliebiges Protokollierungsmodul, das in `stdout` oder `stderr` schreibt. Die aws-lambda-java-core-Bibliothek stellt eine Logger-Klasse mit dem Namen `LambdaLogger` bereit, auf die Sie vom Kontextobjekt zugreifen können. Die Logger-Klasse unterstützt mehrzeilige Protokolle.

Im folgenden Beispiel wird der `LambdaLogger`-Logger verwendet, der vom Kontextobjekt bereitgestellt wird.

Example Handler.java

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example Protokollformat

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
```

```
        ...
    }
  ]
}
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 724 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

Die Java-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details:

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java

Um Ihnen mehr Kontrolle darüber zu geben, wie die Logs Ihrer Funktionen erfasst, verarbeitet und genutzt werden, können Sie die folgenden Protokollierungsoptionen für unterstützte Java-Laufzeiten konfigurieren:

- Protokollformat — Wählen Sie zwischen Klartext und strukturiertem JSON-Format für die Logs Ihrer Funktion

- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, an die Lambda sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Logs sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.

Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Java-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden des strukturierten JSON-Protokollformats mit Java

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die Protokollausgabe unter Verwendung der `LambdaLogger` Klasse als strukturiertes JSON. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"AWSrequestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Abhängig von der verwendeten Protokollierungsmethode können die im JSON-Format erfassten Protokollausgaben Ihrer Funktion auch zusätzliche Schlüssel-Wert-Paare enthalten.

Um Protokollen, die Sie mit dem `LambdaLogger`-Logger erstellen, eine Ebene zuzuweisen, müssen Sie in Ihrem Protokollierungsbefehl ein `LogLevel`-Argument angeben, wie im folgenden Beispiel gezeigt.

Example Protokollierungscode für Java

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

Diese Protokollausgabe durch diesen Beispielcode würde wie folgt in CloudWatch Logs erfasst:

Example JSON-Protokolldatensatz

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

Wenn Sie Ihrer Protokollausgabe kein Level zuweisen, weist Lambda ihr automatisch das Level INFO zu.

Wenn Ihr Code bereits eine andere Logging-Bibliothek verwendet, um strukturierte JSON-Logs zu erstellen, müssen Sie keine Änderungen vornehmen. Lambda codiert keine Protokolle doppelt, die bereits JSON-codiert sind. Selbst wenn Sie Ihre Funktion so konfigurieren, dass sie das JSON-Protokollformat verwendet, erscheinen Ihre Logging-Ausgaben CloudWatch in der von Ihnen definierten JSON-Struktur.

Verwenden der Filterung auf Protokollebene mit Java

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mithilfe der Standard-`LambdaLogger` und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. Lambda filtert dann Ihre Protokollausgaben mithilfe des Schlüsselwertpaars „level“ im JSON-Objekt, wie unter beschrieben. the section called “Verwenden des strukturierten JSON-Protokollformats mit Java” Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unterthe section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.
- Verwenden Sie eine andere Logging-Bibliothek oder Methode, um strukturierte JSON-Logs in Ihrem Code zu erstellen, die ein „Level“ -Schlüsselwertpaar enthalten, das die Ebene der Protokollausgabe definiert. Sie können auch eine beliebige Protokollierungsbibliothek verwenden, die JSON-Protokolle in `stdout` oder `stderr` schreibt. Sie können beispielsweise Powertools for AWS Lambda oder das Paket Log4J2 verwenden, um strukturierte JSON-Protokollausgaben aus Ihrem Code zu generieren. Weitere Informationen dazu finden Sie unter the section called “Verwendung von Powertools für AWS Lambda (Java) und AWS SAM für strukturiertes Logging” und the section called “Implementierung der erweiterten Protokollierung mit Log4j2 und SLF4J”.

Wenn Sie Ihre Funktion so konfigurieren, dass sie Filterung auf Protokollebene verwendet, müssen Sie aus den folgenden Optionen für die Protokollebene auswählen, die Lambda an Logs senden soll: CloudWatch

Protokollebene	Standardnutzung
TRACE (am detailliertesten)	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Detaillierte Informationen für das System-Debugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Meldungen über mögliche Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert
FATAL (am wenigsten Details)	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Lambda sendet Protokolle der ausgewählten Stufe und niedriger bis CloudWatch. Wenn Sie beispielsweise die Protokollebene WARN konfigurieren, sendet Lambda Protokolle, die den Stufen WARN, ERROR und FATAL entsprechen.

Implementierung der erweiterten Protokollierung mit Log4j2 und SLF4J

> **Note**
>
> AWS Lambda nimmt Log4j2 nicht in seine verwalteten Laufzeiten oder Basis-Container-Images auf. Diese sind daher nicht von den unter, und beschriebenen Problemen betroffen. CVE-2021-44228 CVE-2021-45046 CVE-2021-45105
> Für Fälle, in denen eine Kundenfunktion eine betroffene Log4J2-Version enthält, haben wir eine Änderung an den von Lambda Java verwalteten Laufzeiten und Basis-Container-Images vorgenommen, um die Probleme in, und zu beheben. CVE-2021-44228 CVE-2021-45046 CVE-2021-45105 Infolge dieser Änderung sehen Kunden, die Log4J2 verwenden, möglicherweise einen zusätzlichen Protokolleintrag ähnlich wie „`Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)`“. Alle Protokollzeichenfolgen, die in der Log4J2-Ausgabe auf den jndi-Mapper verweisen, werden durch "" ersetzt. `Patched JndiLookup::lookup()`
> Unabhängig von dieser Änderung empfehlen wir allen Kunden, deren Funktionen Log4j2 enthalten, nachdrücklich, auf die neueste Version zu aktualisieren. Insbesondere sollten Kunden, die die Bibliothek aws-lambda-java-log4j2 in ihren Funktionen verwenden, auf Version 1.5.0 (oder höher) aktualisieren und ihre Funktionen erneut bereitstellen. Diese Version aktualisiert die zugrunde liegenden Abhängigkeiten des Log4j2-Dienstprogramms auf Version 2.17.0 (oder höher). Die aktualisierte Binärdatei aws-lambda-java-log4j2 finden Sie unter Maven-Repository und ihren Quellcode finden Sie in Github.
> Beachten Sie abschließend, dass Bibliotheken, die sich auf aws-lambda-java-log4j (v1.0.0 oder 1.0.1) beziehen, unter keinen Umständen verwendet werden sollten. Diese Bibliotheken beziehen sich auf Version 1.x von log4j, deren Nutzungsdauer 2015 abgelaufen ist. Die Bibliotheken werden nicht unterstützt, nicht gewartet, nicht gepatcht und haben bekannte Sicherheitslücken.

Verwenden Sie Apache Log4j2 mit SLF4J, um die Protokollausgabe anzupassen, die Protokollierung bei Komponententests zu unterstützen und AWS SDK-Aufrufe zu protokollieren. Log4j ist eine Logging-Bibliothek für Java-Programme, mit der Sie Protokollebenen konfigurieren und Appender-Bibliotheken verwenden können. SLF4J ist eine Fassadenbibliothek, mit der Sie ändern können, welche Bibliothek Sie verwenden, ohne Ihren Funktionscode zu ändern.

Um die Anforderungs-ID zu den Protokollen Ihrer Funktion hinzuzufügen, verwenden Sie den Appender in der Bibliothek aws-lambda-java-log4j2 .

Example src/main/.xml resources/log4j2 — Appender-Konfiguration

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
       <AppenderRef ref="Lambda"/>
     </Root>
     <Logger name="software.amazon.awssdk" level="WARN" />
     <Logger name="software.amazon.awssdk.request" level="DEBUG" />
   </Loggers>
 </Configuration>
```

Sie können entscheiden, wie Ihre Log4j2-Protokolle entweder für Klartext- oder JSON-Ausgaben konfiguriert werden sollen, indem Sie unter den Tags `<LambdaTextFormat>` und `<LambdaJSONFormat>` ein Layout angeben.

In diesem Beispiel wird im Textmodus jeder Zeile das Datum, die Uhrzeit, die Anforderungs-ID, die Protokollstufe und der Klassenname vorangestellt. Im JSON-Modus wird das `<JsonTemplateLayout>` mit einer Konfiguration verwendet, die zusammen mit der `aws-lambda-java-log4j2`-Bibliothek geliefert wird.

SLF4J ist eine Fassadenbibliothek zum Protokollieren in Java-Code. In Ihrem Funktionscode verwenden Sie die SLF4J-Logger-Factory, um einen Logger mit Methoden für Protokollstufen wie `info()` und `warn()` abzurufen. Bei Ihrer Build-Konfiguration schließen Sie die Protokollierungsbibliothek und den SLF4J-Adapter in den Klassenpfad ein. Durch Ändern der Bibliotheken in der Build-Konfiguration können Sie den Logger-Typ ändern, ohne den Funktionscode zu ändern. SLF4J ist erforderlich, um Protokolle aus SDK for Java zu erfassen.

Im folgenden Beispielcode verwendet die Handler-Klasse SLF4J, um einen Logger abzurufen.

Example src/main/java/example/HandlerS3.java — Protokollierung mit SLF4J

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
}
```

Es wird eine ähnliche Protokollausgabe wie die folgende erstellt:

Example Protokollformat

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
```

```
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

Die Build-Konfiguration nimmt Laufzeitabhängigkeiten auf dem Lambda-Appender sowie SLF4J-Adapter und Implementierungsabhängigkeiten auf Log4j2 an.

Example build.gradle – Protokollierungsabhängigkeiten

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

Wenn Sie Ihren Code vor Ort für Tests ausführen, ist das Kontextobjekt mit dem Lambda-Logger nicht verfügbar, und es gibt keine Anforderungs-ID, die der Lambda-Appender verwenden kann. Beispiele für Testkonfigurationen finden Sie in den Beispielanwendungen im nächsten Abschnitt.

Andere Tools und Bibliotheken für die Protokollierung verwenden

Powertools for AWS Lambda (Java) ist ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit. Das Logging-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen zum Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)
- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu

- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (Java) und AWS SAM für strukturiertes Logging

Gehen Sie wie folgt vor, um eine Hello World Java-Beispielanwendung mit integrierten Powertools for AWS Lambda (Java) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den AWS SAM. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java 11
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mit der Hello World Java-Vorlage.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

   > **Note**
   >
   > Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

   ```
   aws cloudformation describe-stacks --stack-name sam-app --query
    'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
   ```

6. Rufen Sie den API-Endpunkt auf:

   ```
   curl -X GET <URL_FROM_PREVIOUS_STEP>
   ```

   Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

   ```
   {"message":"hello world"}
   ```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

   ```
   sam logs --stack-name sam-app
   ```

   Das Ergebnis sieht folgendermaßen aus:

   ```
   2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
    INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
   central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
   2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
    Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
   2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
    Transforming org/apache/logging/log4j/core/lookup/JndiLookup
    (lambdainternal.CustomerClassLoader@1a6c5a9e)
   2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
    START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
   2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
   ```

```
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
```

```
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 5275 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Logs in der CloudWatch Konsole anzeigen

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Logs für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-`

`in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Beispielprotokolliercode

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung verschiedener Protokollierungskonfigurationen demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine AWS SAM Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java — Eine Java-Event-Funktion, die demonstriert, wie Sie Lambda zur Bearbeitung von Bestellungen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das SDK verwenden und wie Sie die AWS Ausgabe protokollieren.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Die `java-basic`-Beispielanwendung zeigt eine minimale Protokollierungskonfiguration, die Protokollierungstests unterstützt. Der Handler-Code verwendet den `LambdaLogger`-Logger, der vom Kontextobjekt bereitgestellt wird. Für Tests verwendet die Anwendung eine benutzerdefinierte `TestLogger`-Klasse, die die `LambdaLogger`-Schnittstelle mit einem Log4j2-Logger implementiert. Es verwendet SLF4J als Fassade für die Kompatibilität mit dem SDK. AWS Protokollbibliotheken werden von der Build-Ausgabe ausgeschlossen, um das Bereitstellungspaket klein zu halten.

Instrumentierung von Java-Code in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können es verwenden X-Ray, um eine Anfrage zu verfolgen, während sie Ressourcen in Ihrer Anwendung durchquert, zu denen Lambda-Funktionen und andere AWS Dienste gehören können.

Um Ablaufverfolgungsdaten an zu senden X-Ray, können Sie eine von zwei SDK-Bibliotheken verwenden:

- AWS Distribution for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des (OTel) SDK. OpenTelemetry
- AWS X-Ray SDK für Java— Ein SDK zum Generieren und Senden von Trace-Daten an. X-Ray
- Powertools for AWS Lambda (Java) — Ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den Dienst zu senden. X-Ray Anschließend können Sie X-Ray die Leistungskennzahlen Ihrer Anwendung anzeigen, filtern und Einblicke in sie gewinnen, um Probleme und Optimierungsmöglichkeiten zu identifizieren.

> **Important**
>
> Die SDKs X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von angeboten wird. AWS Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgängige Ablaufverfolgung implementieren. X-Ray Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Auswahl zwischen der AWS Distribution for Open Telemetry und SDKs. X-Ray

Sections

- Verwenden von Powertools für AWS Lambda (Java) und AWS SAM für die Rückverfolgung
- Verwenden von Powertools für AWS Lambda (Java) und die AWS CDK für die Rückverfolgung
- Verwenden von ADOT zur Instrumentierung Ihrer Java-Funktionen
- Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Java-Funktionen

- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Die Ablaufverfolgung wird aktiviert mit CloudFormation
- Interpretieren und X-Ray Trace
- Speichern von Laufzeitabhängigkeiten in einer Ebene (X-Ray SDK)
- X-Ray Ablaufverfolgung in Beispielanwendungen (SDK) X-Ray

Verwenden von Powertools für AWS Lambda (Java) und AWS SAM für die Rückverfolgung

Gehen Sie wie folgt vor, um eine Hello World Java-Beispielanwendung mit integrierten Powertools for AWS Lambda (Java) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java 11 oder höher
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mit der Hello World Java-Vorlage.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
New XRay Service Graph
  Start time: 2025-02-03 14:31:48+01:00
  End time: 2025-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
```

```
    - fault count(5XX): 0
    - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 3] at (2025-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwenden von Powertools für AWS Lambda (Java) und die AWS CDK für die Rückverfolgung

Gehen Sie wie folgt vor, um eine Hello World Java-Beispielanwendung mit integrierten Powertools for AWS Lambda (Java) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den. AWS CDK Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon API Gateway Gateway-Endpunkt und einer Lambda Event Function. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java 11 oder höher

- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS CDK Anwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language java
```

3. Erstellen Sie ein Maven-Projekt mit dem folgenden Befehl:

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. Öffnen Sie `pom.xml` im `hello-world\app\Function`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code, der Abhängigkeiten und Maven-Plugins für Powertools enthält.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
```

```
    <log4j.version>2.17.2</log4j.version>
</properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
```

```
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>
```

```
                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
                </dependencies>
        </plugin>
    </plugins>
</build>
</project>
```

5. Erstellen Sie das `hello-world\app\src\main\resource`-Verzeichnis und erstellen Sie `log4j.xml` für die Protokollkonfiguration.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Öffnen Sie `log4j.xml` und fügen Sie den folgenden Code hinzu.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. Öffnen Sie `App.java` aus dem `hello-world\app\Function\src\main\java\helloworld`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code. Dies ist der Code für die Lambda-Funktion.

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");
```

```
        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. Öffnen Sie `HelloWorldStack.java` aus dem `hello-world\src\main\java\com\myorg`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code. Dieser Code verwendet den Lambda-Konstruktor und den ApiGatewayv2-Konstruktor, um eine REST-API und eine Lambda-Ereignisfunktion zu erstellen.

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
```

```
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
 dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
 "/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
```

```
                .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
                .build());

        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
 HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. Öffnen Sie `pom.xml` aus dem `hello-world`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
```

```
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
```

```
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. Stellen Sie sicher, dass Sie sich im `hello-world` Verzeichnis befinden, und stellen Sie Ihre Anwendung bereit.

```
cdk deploy
```

11. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

13. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
New XRay Service Graph
  Start time: 2025-02-03 14:59:50+00:00
  End time: 2025-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2025-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Verwenden von ADOT zur Instrumentierung Ihrer Java-Funktionen

ADOT bietet vollständig verwaltete Lambda-Ebenen, die alles packen, was Sie zum Sammeln von Telemetriedaten mit dem OTel-SDK benötigen. Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Ebenen auch für die benutzerdefinierte Initialisierung von OTel konfigurieren. Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Java-Laufzeiten können Sie zwischen zwei Ebenen wählen, die verbraucht werden sollen:

- AWS verwaltete Lambda-Schicht für ADOT Java (Auto-instrumentation Agent) — Diese Schicht transformiert Ihren Funktionscode beim Start automatisch, um Tracing-Daten zu sammeln. Eine ausführliche Anleitung, wie Sie diese Ebene zusammen mit dem ADOT-Java-Agenten verwenden können, finden Sie in der ADOT-Dokumentation unter AWS Distro for OpenTelemetry Lambda Support for Java (Auto-instrumentation Agent).
- AWS verwaltete Lambda-Schicht für ADOT Java — Diese Schicht bietet auch eine integrierte Instrumentierung für Lambda-Funktionen, erfordert jedoch einige manuelle Codeänderungen, um das OTel SDK zu initialisieren. Eine ausführliche Anleitung zur Nutzung dieser Ebene finden Sie unter AWS Distro for OpenTelemetry Lambda Support for Java in der ADOT-Dokumentation.

Anweisungen zur Konvertierung von Lambda-Telemetrie-API-Ereignissen in OpenTelemetry Spans finden Sie unter. the section called “Konvertieren von Ereignissen in OTel-Spans”

Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Java-Funktionen

Um Daten über Aufrufe aufzuzeichnen, die Ihre Funktion an andere Ressourcen und Dienste in Ihrer Anwendung tätigt, können Sie das X-Ray SDK for Java zu Ihrer Build-Konfiguration hinzufügen. Das folgende Beispiel zeigt eine Gradle-Build-Konfiguration, die die Bibliotheken enthält, die die automatische Instrumentierung von AWS SDK for Java 2.x Clients aktivieren.

Example build.gradle — Abhängigkeiten verfolgen

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

Nachdem Sie die richtigen Abhängigkeiten hinzugefügt und die erforderlichen Codeänderungen vorgenommen haben, aktivieren Sie die Ablaufverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration

- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpretieren und X-Ray Trace

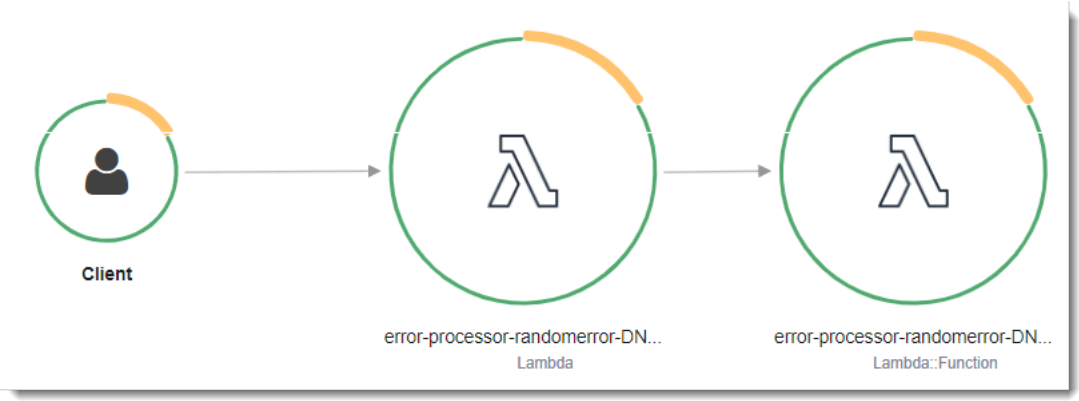
Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen X-Ray. Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray Dienstdiagramm zeigt Informationen zu Ihrer Anwendung und all ihren Komponenten. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK X-Ray, um Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

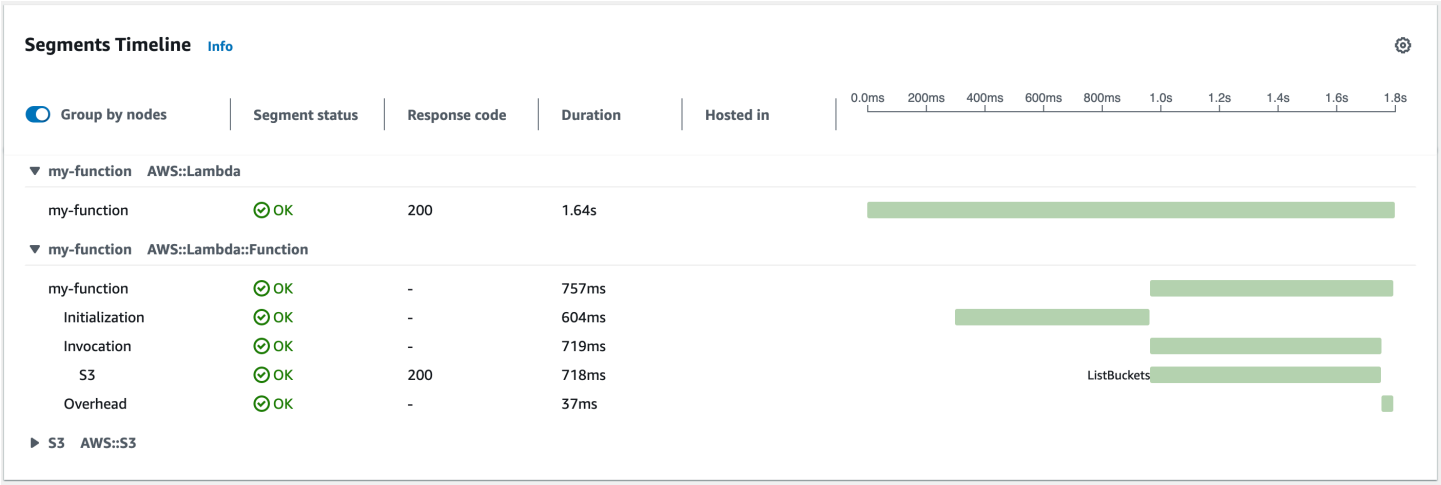


X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

In X-Ray zeichnet ein Trace Informationen über eine Anfrage auf, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen stellen Sie möglicherweise geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollmeldungen und Trace-Segmenten fest, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.

Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Note

Zu den SnapStart-Funktionen gehört auch ein `Restore`-Teilsegment. Das Teilsegment `Restore` zeigt die Zeit an, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und vorhandene Laufzeit-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird im `Restore`-Untersegment erfasst. Die Zeit, die Sie außerhalb der microVM für die Wiederherstellung eines Snapshots aufwenden, wird Ihnen nicht in Rechnung gestellt.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray SDK für Java im AWS X-Ray -Entwicklerhandbuch.

> **Preisgestaltung**
>
> Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit kostenlos nutzen. AWS Bei Überschreitung dieses Schwellenwerts X-Ray fallen Gebühren für die Speicherung und den Abruf von Trace an. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Speichern von Laufzeitabhängigkeiten in einer Ebene (X-Ray SDK)

Wenn Sie das X-Ray SDK verwenden, um AWS SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich umfangreich werden. Um zu vermeiden, dass bei jeder Aktualisierung Ihres Funktionscodes Laufzeitabhängigkeiten hochgeladen werden, packen Sie das X-Ray SDK in eine Lambda-Schicht.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion` Ressource, die das AWS SDK für Java und X-Ray SDK for Java speichert.

Example template.yml — Ebene „Abhängigkeiten“

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java25
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Änderungen an der Build-Konfiguration, um das Layer-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Java-Basic-Beispielanwendung unter. GitHub

X-Ray Ablaufverfolgung in Beispielanwendungen (SDK) X-Ray

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung von X-Ray Tracing demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine AWS SAM Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java — Eine Java-Event-Funktion, die demonstriert, wie Sie Lambda zur Bearbeitung von Bestellungen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das SDK verwenden und wie Sie die AWS Ausgabe protokollieren.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Für alle Beispielanwendungen ist die aktive Ablaufverfolgung für Lambda-Funktionen aktiviert. Die `s3-java` Anwendung zeigt beispielsweise die automatische Instrumentierung von AWS SDK for Java 2.x Clients, Segmentmanagement für Tests, benutzerdefinierte Untersegmente und die Verwendung von Lambda-Schichten zum Speichern von Laufzeitabhängigkeiten.

Java-Beispielanwendungen für AWS Lambda

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung von Java in demonstrieren AWS Lambda. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine CloudFormation Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java — Eine Java-Event-Funktion, die demonstriert, wie Sie Lambda zur Bearbeitung von Bestellungen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das SDK verwenden und wie Sie die AWS Ausgabe protokollieren.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Ausführen beliebter Java-Frameworks auf Lambda

- spring-cloud-function-samples — Ein Beispiel aus Spring, das zeigt, wie das Spring Cloud Function-Framework verwendet wird, um Lambda Event Functions zu erstellen. AWS
- Serverlose Spring Boot-Anwendungsdemo — Ein Beispiel, das zeigt, wie eine typische Spring Boot-Anwendung in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit eingerichtet wird.
- Serverlose Micronaut-Anwendungsdemo — Ein Beispiel, das zeigt, wie Micronaut in einer verwalteten Java-Laufzeit mit und ohne oder als natives SnapStart GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet wird. Erfahren Sie mehr in den Leitfäden. Micronaut/Lambda

- Serverlose Quarkus-Anwendungsdemo — Ein Beispiel, das zeigt, wie Quarkus in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet wird. Weitere Informationen finden Sie in der Anleitung und im Leitfaden. Quarkus/Lambda Quarkus/SnapStart

Wenn Sie mit Lambda Event Functions in Java noch nicht vertraut sind, beginnen Sie mit den `java-basic` Beispielen. Für erste Schritte mit Lambda-Ereignisquellen schauen Sie sich die `java-events`-Beispiele an. Beide Beispielsätze zeigen die Verwendung der Java-Bibliotheken, Umgebungsvariablen, des SDK und des AWS AWS X-Ray SDK von Lambda. Diese Beispiele erfordern nur eine minimale Einrichtung. Sie können sie in weniger als einer Minute von der Befehlszeile aus bereitstellen.

Erstellen von Lambda-Eventfunktionen mit Go

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie eine OS-only Laufzeit (die `provided` Runtime-Familie), um Go Event Functions auf Lambda bereitzustellen.

Hinweise zur Verwendung von Ebenen mit Go finden Sie unter the section called “Layer”.

Themen

- Unterstützte Go-Laufzeiten
- Tools und Bibliotheken
- Definieren Sie Lambda-Ereignisfunktionshandler in Go
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Go-Event-Funktion
- Stellen Sie Go Lambda Event Functions mit.zip-Dateiarchiven bereit
- Stellen Sie Go Lambda Event Functions mit Container-Images bereit
- Arbeiten mit Ebenen für Go Lambda Event Functions
- Go Lambda Event Functions protokollieren und überwachen
- Go-Code instrumentieren AWS Lambda

Unterstützte Go-Laufzeiten

Die verwaltete Laufzeit Go 1.x für Lambda ist veraltet. Wenn Sie Funktionen haben, die die Go 1.x-Laufzeit verwenden, müssen Sie Ihre Funktionen nach `provided.al2023` oder `provided.al2`

migrieren. Das `provided.al2023` und `provided.al2` Laufzeiten bieten mehrere Vorteile gegenüber `go1.x`, einschließlich der Unterstützung der arm64-Architektur (AWS Graviton-Prozessoren), kleinere Binärdateien und etwas schnellere Aufrufzeiten.

Für diese Migration sind keine Codeänderungen erforderlich. Die einzigen erforderlichen Änderungen betreffen die Erstellung Ihres Bereitstellungspakets sowie die Laufzeit, die Sie zur Erstellung Ihrer Funktion verwenden. Weitere Informationen finden Sie unter Migrieren AWS Lambda Funktionen aus dem Go1.x Laufzeit zur benutzerdefinierten Laufzeit auf Amazon Linux 2 auf der AWS Compute-Blog.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
OS-only Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029

Tools und Bibliotheken

Lambda stellt die folgenden Tools und Bibliotheken für die Go-Laufzeit bereit:

- AWS SDK für Go v2: Der Offizielle AWS SDK für die Go-Programmiersprache.
- Github.com/aws/aws-lambda-go/lambda: Die Implementierung des Lambda-Programmiermodells für Go. Dieses Paket wird verwendet von AWS Lambda um deine aufzurufen Handler.
- Github.com/aws/aws-lambda-go/lambdacontext: Hilfsprogramme für den Zugriff auf Kontextinformationen aus dem Kontextobjekt.
- Github.com/aws/aws-lambda-go/events: Diese Bibliothek stellt Typdefinitionen für gängige Integrationen von Ereignisquellen bereit.
- Github.com/aws/aws-lambda-go/cmd/build-lambda-zip: Dieses Tool kann verwendet werden, um ein ZIP-Dateiarchiv unter Windows zu erstellen.

Weitere Informationen finden Sie unter aws-lambda-Go auf GitHub.

Lambda stellt die folgenden Beispielanwendungen für die Go-Laufzeit bereit:

Lambda-Beispielanwendungen in Go

- go-al2: Eine Hello World-Funktion, die die öffentliche IP-Adresse zurückgibt. Diese App verwendet die benutzerdefinierte Laufzeit `provided.al2`.

- leerlos — Eine Go-Funktion, die die Verwendung der Go-Bibliotheken, der Protokollierung, der Umgebungsvariablen und der AWS SDK. Diese App verwendet die Laufzeit `go1.x`.

Definieren Sie Lambda-Ereignisfunktionshandler in Go

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda Event Function-Handlern in Go arbeiten, einschließlich der Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Go-Lambda-Funktion, die Informationen über einen Auftrag aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einem Bucket von Amazon Simple Storage Service (Amazon S3) platziert. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Themen

- Einrichten Ihres Go-Handler-Projekts
- Beispiel für einen Go Lambda Event Function Code
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Gültige Handler-Signaturen für Go-Handler
- Verwendung der AWS SDK für Go v2 in deinem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für Go Lambda Event Functions

Einrichten Ihres Go-Handler-Projekts

Eine Lambda-Funktion in Go wird als ausführbare Go-Datei erstellt. Sie können ein Go-Lambda-Funktionsprojekt auf die gleiche Weise initialisieren wie jedes andere Go-Projekt, indem Sie den folgenden `go mod init`-Befehl verwenden:

```
go mod init example-go
```

Hier ist `example-go` der Name des Moduls. Sie können dies durch alles ersetzen. Dieser Befehl initialisiert Ihr Projekt und erzeugt die `go.mod`-Datei, in der die Abhängigkeiten Ihres Projekts aufgelistet sind.

Verwenden Sie den Befehl `go get`, um externe Abhängigkeiten zu Ihrem Projekt hinzuzufügen. Beispielsweise müssen Sie für alle Lambda-Funktionen in Go den Github einbeziehen. com/awsgo/lambda/aws-lambda--Paket, das das Lambda-Programmiermodell für Go implementiert. Binden Sie dieses Paket mit dem folgenden `go get`-Befehl ein:

```
go get github.com/aws/aws-lambda-go
```

Ihr Funktionscode sollte sich in einer Go-Datei befinden. Im folgenden Beispiel benennen wir diese Datei `main.go`. In dieser Datei implementieren Sie Ihre Kernfunktionslogik in einer Handler-Methode sowie in einer `main()`-Funktion, die diesen Handler aufruft.

Beispiel für einen Go Lambda Event Function Code

Das folgende Beispiel für einen Go-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`main.go` Lambda-Ereignisfunktion**

```
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "log"
 "os"
 "strings"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

type Order struct {
 OrderID string  `json:"order_id"`
 Amount  float64 `json:"amount"`
 Item    string  `json:"item"`
}
```

```
var (
 s3Client *s3.Client
)

func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("unable to load SDK config, %v", err)
 }

 s3Client = s3.NewFromConfig(cfg)
}

func uploadReceiptToS3(ctx context.Context, bucketName, key, receiptContent string)
 error {
 _, err := s3Client.PutObject(ctx, &s3.PutObjectInput{
  Bucket: &bucketName,
  Key:    &key,
  Body:   strings.NewReader(receiptContent),
 })
 if err != nil {
  log.Printf("Failed to upload receipt to S3: %v", err)
  return err
 }
 return nil
}

func handleRequest(ctx context.Context, event json.RawMessage) error {
 // Parse the input event
 var order Order
 if err := json.Unmarshal(event, &order); err != nil {
  log.Printf("Failed to unmarshal event: %v", err)
  return err
 }

 // Access environment variables
 bucketName := os.Getenv("RECEIPT_BUCKET")
 if bucketName == "" {
  log.Printf("RECEIPT_BUCKET environment variable is not set")
  return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
 }
```

```
 // Create the receipt content and key destination
 receiptContent := fmt.Sprintf("OrderID: %s\nAmount: $%.2f\nItem: %s",
  order.OrderID, order.Amount, order.Item)
 key := "receipts/" + order.OrderID + ".txt"

 // Upload the receipt to S3 using the helper method
 if err := uploadReceiptToS3(ctx, bucketName, key, receiptContent); err != nil {
  return err
 }

 log.Printf("Successfully processed order %s and stored receipt in S3 bucket %s",
 order.OrderID, bucketName)
 return nil
}

func main() {
 lambda.Start(handleRequest)
}
```

Diese `main.go`-Datei enthält die folgenden Abschnitte des Codes:

- `package main`: In Go muss das Paket, das Ihre `func main()`-Funktion enthält, immer `main` heißen.
- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt.
- `type Order struct {}`-Block: Definieren Sie die Form des erwarteten Eingabeereignisses in dieser Go-Struktur.
- `var ()`-Block: Verwenden Sie diesen Block, um alle globalen Variablen zu definieren, die Sie in Ihrer Lambda-Funktion verwenden werden.
- `func init() {}`: Fügen Sie in diese `init()`-Methode beliebigen Code ein, den Lambda während der Initialisierungsphase ausführen soll.
- `func uploadReceiptToS3(...) {}`: Dies ist eine Hilfsmethode, auf die von der `handleRequest`-Haupthandler-Methode verwiesen wird.
- `func handleRequest(ctx context.Context, event json.RawMessage) error {}`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `func main() {}`: Dies ist ein erforderlicher Einstiegspunkt für Ihren Lambda-Handler. Das Argument für die `lambda.Start()`-Methode ist Ihre Haupthandler-Methode.

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

Namenskonventionen für Handler

Für Lambda-Funktionen in Go können Sie einen beliebigen Namen für den Handler verwenden. In diesem Beispiel lautet der Name der Handler-Methode `handleRequest`. Sie können die Umgebungsvariable `_HANDLER` verwenden, um in Ihrem Code auf den Handler-Wert zu verweisen.

Bei Go-Funktionen, die mit einem .zip-Bereitstellungspaket bereitgestellt werden, muss die ausführbare Datei, die Ihren Funktionscode enthält, den Namen `bootstrap` tragen. Darüber hinaus muss sich die `bootstrap`-Datei im Stammverzeichnis der .zip-Datei befinden. Für Go-Funktionen, die über ein Container-Image bereitgestellt werden, können Sie einen beliebigen Namen für die ausführbare Datei verwenden.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in Go arbeiten, können Sie die Form des erwarteten Eingabeereignisses als Go-Struktur definieren. In diesem Beispiel definieren wir eine Struktur, die einen `Order` darstellt:

```
type Order struct {
    OrderID string  `json:"order_id"`
    Amount  float64 `json:"amount"`
    Item    string  `json:"item"`
}
```

Diese Struktur entspricht der erwarteten Eingabeform. Nachdem Sie Ihre Struktur definiert haben, können Sie eine Handler-Signatur schreiben, die einen generischen JSON-Typ verwendet, der mit der encoding/json Standardbibliothek kompatibel ist. Sie können es dann mit der Funktion

func Unmarshal in Ihre Struktur deserialisieren. Dies wird in den ersten Zeilen des Handlers veranschaulicht:

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    // Parse the input event
    var order Order
    if err := json.Unmarshal(event, &order); err != nil {
        log.Printf("Failed to unmarshal event: %v", err)
        return err
    ...
}
```

Nach dieser Deserialisierung können Sie auf die Felder der `order`-Variablen zugreifen. Zum Beispiel holt `order.OrderID` den Wert von `"order_id"` aus der ursprünglichen Eingabe.

> **Note**
>
> Das `encoding/json`-Paket kann nur auf exportierte Felder zugreifen. Um exportiert werden, müssen Feldnamen in der Ereignisstruktur großgeschrieben werden.

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In diesem Beispiel haben wir diese Variable wie `ctx` in der Handler-Signatur deklariert:

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    ...
}
```

Die `ctx context.Context`-Eingabe ist ein optionales Argument in Ihrem Funktionshandler. Weitere Informationen zu akzeptierten Handler-Signaturen finden Sie unter the section called “Gültige Handler-Signaturen für Go-Handler”.

Wenn Sie mithilfe des AWS SDK andere Dienste aufrufen, ist das Kontextobjekt in einigen wichtigen Bereichen erforderlich. Um beispielsweise Ihre SDK-Clients ordnungsgemäß zu initialisieren, können Sie die richtige AWS -SDK-Konfiguration mithilfe des Kontextobjekts wie folgt laden:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
```

SDK-Aufrufe selbst erfordern möglicherweise das Kontextobjekt als Eingabe. Der Aufruf von `s3Client.PutObject` akzeptiert beispielsweise das Kontextobjekt als erstes Argument:

```
// Upload the receipt to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

Außerhalb von AWS SDK-Anfragen können Sie das Kontextobjekt auch für die Funktionsüberwachung verwenden. Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Kontext”.

Gültige Handler-Signaturen für Go-Handler

Sie haben bei der Erstellung eines Lambda-Funktions-Handlers in Go mehrere Möglichkeiten. Allerdings müssen die folgenden Regeln beachten:

- Der Handler muss eine Funktion sein.
- Der Handler kann zwischen 0 und 2 Argumente annehmen. Bei zwei Argumenten muss das erste Argument implementiere `context.Context`.
- Der Handler kann zwischen 0 und 2 Argumente zurückgeben. Bei einem einzigen Rückgabewert muss er implementiere `error`. Bei zwei Rückgabewerten muss der zweite Wert implementiere `error`.

Im Folgenden werden gültige Handler-Signaturen aufgeführt. `TInund TOut` stellen Typen dar, die mit der encoding/jsonStandardbibliothek kompatibel sind. Weitere Informationen zur Deserialisierung dieser Typen finden Sie unter func Unmarshal.

- `func ()`
- `func () error`
- `func () (TOut, error)`
- `func (TIn) error`
- `func (TIn) (TOut, error)`

- `func (context.Context) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) error`
- `func (context.Context, TIn) (TOut, error)`

Verwendung der AWS SDK für Go v2 in deinem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für Go v2.

> **Note**
>
> Das AWS SDK für Go (v1) befindet sich im Wartungsmodus und der Support endet am 31. Juli 2025. Wir empfehlen, dass Sie in Zukunft nur noch AWS SDK für Go Version 2 verwenden.

Um Ihrer Funktion SDK-Abhängigkeiten hinzuzufügen, verwenden Sie den `go get`-Befehl für die spezifischen SDK-Clients, die Sie benötigen. Im Beispielcode zuvor haben wir die `config`-Bibliothek und die `s3`-Bibliothek verwendet. Fügen Sie diese Abhängigkeiten hinzu, indem Sie die folgenden Befehle in dem Verzeichnis ausführen, das Ihre `go.mod`- und `main.go` -Dateien enthält:

```
go get github.com/aws/aws-sdk-go-v2/config
go get github.com/aws/aws-sdk-go-v2/service/s3
```

Importieren Sie dann die Abhängigkeiten entsprechend in den Importblock Ihrer Funktion:

```
import (
    ...
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)
```

Wenn Sie das SDK in Ihrem Handler verwenden, konfigurieren Sie Ihre Clients mit den richtigen Einstellungen. Der einfachste Weg, dies zu tun, besteht darin, die standardmäßige Anbieterkette für Anmeldeinformationen zu verwenden. Dieses Beispiel zeigt eine Möglichkeit, diese Konfiguration zu laden:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        log.Printf("Failed to load AWS SDK config: %v", err)
        return err
    }
```

Nachdem Sie diese Konfiguration in die `cfg`-Variable geladen haben, können Sie diese Variable an Client-Instanziierungen übergeben. Der Beispielcode erstellt einen Amazon S3 S3-Client wie folgt:

```
// Create an S3 client
    s3Client := s3.NewFromConfig(cfg)
```

In diesem Beispiel haben wir unseren Amazon-S3-Client in der `init()`-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Das Problem ist, dass Lambda in der `init()`-Funktion keinen Zugriff auf das Kontextobjekt hat. Als Abhilfe können Sie in der Initialisierungsphase einen Platzhalter wie `context.TODO()` übergeben. Wenn Sie später über den Client einen Anruf tätigen, übergeben Sie das vollständige Kontextobjekt. Diese Abhilfemaßnahme wird auch in the section called “Verwenden Sie den Kontext in AWS Initialisierungen und Aufrufe des SDK-Clients” beschrieben.

Nachdem Sie Ihren SDK-Client konfiguriert und initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS -Diensten verwenden. Der Beispielcode ruft die Amazon S3 `PutObject`-API wie folgt auf:

```
_, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
    Bucket: &bucketName,
    Key:    &key,
    Body:   strings.NewReader(receiptContent),
})
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `os.Getenv()`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
// Access environment variables
    bucketName := os.Getenv("RECEIPT_BUCKET")
    if bucketName == "" {
        log.Printf("RECEIPT_BUCKET environment variable is not set")
        return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
    }
```

Verwenden des globalen Zustands

Um zu vermeiden, dass bei jedem Funktionsaufruf neue Ressourcen erstellt werden, können Sie globale Variablen außerhalb des Handler-Codes Ihrer Lambda-Funktion deklarieren und ändern. Sie definieren diese globalen Variablen in einem `var`-Block oder einer Anweisung. Darüber hinaus kann Ihr Handler eine `init()` Funktion deklarieren, die während der Initialisierungsphase ausgeführt wird. Die `init` Methode verhält sich genauso AWS Lambda wie in Standard-Go-Programmen.

Bewährte Codemethoden für Go Lambda Event Functions

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die Best Practices für Codierung anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Mit diesem Ansatz können Sie eine Funktion erstellen, die besser anhand von Einheiten getestet werden kann.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch wird die Zeit reduziert, die das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates auf die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Go-Event-Funktion

In Lambda bietet das Kontextobjekt Methoden und Eigenschaften mit Informationen über den Aufruf, die Funktion und die Ausführungsumgebung. Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler übergeben. Um das Kontextobjekt in Ihrem Handler zu verwenden, können Sie es optional als Eingabeparameter für Ihren Handler deklarieren. Das Kontextmenüobjekt ist erforderlich, wenn Sie in Ihrem Handler Folgendes tun möchten:

- Sie benötigen Zugriff auf eine der globalen Variablen, Methoden oder Eigenschaften, die vom Kontextobjekt angeboten werden. Diese Methoden und Eigenschaften sind nützlich für Aufgaben wie die Bestimmung der Entität, die Ihre Funktion aufgerufen hat, oder die Messung der Aufrufzeit Ihrer Funktion, wie in the section called “Zugreifen auf Aufrufkontextinformationen” dargestellt.
- Sie müssen den verwenden, AWS SDK für Go um Anrufe an andere Dienste zu tätigen. Das Kontextobjekt ist ein wichtiger Eingabeparameter für die meisten dieser Aufrufe. Weitere Informationen finden Sie unter the section called “Verwenden Sie den Kontext in AWS Initialisierungen und Aufrufe des SDK-Clients”.

Themen

- Unterstützte Variablen, Methoden und Eigenschaften im Kontextmenüobjekt
- Zugreifen auf Aufrufkontextinformationen
- Verwenden Sie den Kontext in AWS Initialisierungen und Aufrufe des SDK-Clients

Unterstützte Variablen, Methoden und Eigenschaften im Kontextmenüobjekt

Die Lambda-Kontextbibliothek bietet die folgenden globalen Variablen, Methoden und Eigenschaften.

Globale Variablen

- `FunctionName` – Der Name der Lambda-Funktion.
- `FunctionVersion` – Die Version der Funktion.
- `MemoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `LogGroupName` – Protokollgruppe für die Funktion.
- `LogStreamName` – Der Protokollstrom für die Funktionsinstance.

Context-Methoden

- `Deadline` – Gibt das Datum zurück, an dem eine Zeitüberschreitung für die Ausführung eintritt (in Unix-Millisekunden).

Context-Eigenschaften

- `InvokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `AwsRequestID` – Der Bezeichner der Aufrufanforderung.
- `Identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `ClientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.

Zugreifen auf Aufrufkontextinformationen

Lambda-Funktionen haben Zugriff auf Metadaten über ihre Umgebung und die Aufrufanforderung. Darauf kann unter Paketkontext zugegriffen werden. Falls Ihr Handler `context.Context` als Parameter umfasst, fügt Lambda Informationen über Ihre Funktion in der `Value`-Eigenschaft des Kontexts ein. Beachten Sie, dass Sie die `lambdacontext`-Bibliothek importieren müssen, um auf die Inhalte des `context.Context`-Objekts zuzugreifen.

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
        lambda.Start(CognitoHandler)
}
```

Im obigen Beispiel `lc` wird die Variable verwendet, um die Informationen zu verarbeiten, die das Kontextobjekt erfasst hat, und `log.Print(lc.Identity.CognitoIdentityPoolID)` druckt diese Informationen, in diesem Fall die CognitoIdentityPoolID.

Das folgende Beispiel bietet eine Einführung in die Verwendung der Kontextobjekte zur Überwachung der Abschlussdauer Ihrer Lambda-Funktion. Auf diese Weise können Sie die Leistungserwartungen analysieren und Ihren Funktionscode bei Bedarf entsprechend anpassen.

```
package main

import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil

                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
}
```

Verwenden Sie den Kontext in AWS Initialisierungen und Aufrufe des SDK-Clients

Wenn Ihr Handler das verwenden muss, um Aufrufe AWS SDK für Go an andere Dienste zu tätigen, fügen Sie das Kontextobjekt als Eingabe für Ihren Handler hinzu. In ist es eine bewährte Methode AWS, das Kontextobjekt bei den meisten AWS SDK-Aufrufen zu übergeben. Der Aufruf von Amazon S3 `PutObject` akzeptiert beispielsweise das Kontextobjekt (`ctx`) als erstes Argument:

```
// Upload an object to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

Um Ihre SDK-Clients richtig zu initialisieren, können Sie auch das Kontext-Objekt verwenden, um die richtige Konfiguration zu laden, bevor Sie dieses Konfigurationsobjekt an den Client übergeben:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    ...
    s3Client = s3.NewFromConfig(cfg)
```

Wenn Sie Ihre SDK-Clients außerhalb Ihres Haupthandlers initialisieren möchten (d. h. während der Initialisierungsphase), können Sie ein Platzhalter-Kontextobjekt übergeben:

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 ...
 s3Client = s3.NewFromConfig(cfg)
}
```

Wenn Sie Ihre Clients auf diese Weise initialisieren, stellen Sie sicher, dass Sie bei SDK-Aufrufen von Ihrem Haupthandler das richtige Kontextobjekt übergeben.

Stellen Sie Go Lambda Event Functions mit.zip-Dateiarchiven bereit

Der Code Ihrer AWS Lambda Funktion besteht aus Skripten oder kompilierten Programmen und deren Abhängigkeiten. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und ZIP-Dateiarchiven.

Auf dieser Seite wird beschrieben, wie Sie eine .zip-Datei als Bereitstellungspaket für die Go-Laufzeit erstellen und dann die .zip-Datei verwenden, um Ihren Funktionscode AWS Lambda mithilfe von AWS-Managementkonsole, AWS Command Line Interface (AWS CLI) und AWS Serverless Application Model (AWS SAM) bereitzustellen.

Beachten Sie, dass Lambda POSIX-Dateiberechtigungen verwendet, sodass Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen müssen, bevor Sie das ZIP-Dateiarchiv erstellen.

Sections

- Erstellen einer ZIP-Datei unter macOS und Linux
- Erstellen einer ZIP-Datei unter Windows
- Go Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren

Erstellen einer ZIP-Datei unter macOS und Linux

Im Folgenden wird beschrieben, wie Sie Ihre ausführbare Datei mithilfe des Befehls `go build` kompilieren und ein Bereitstellungspaket in Form einer ZIP-Datei für Lambda erstellen. Stellen Sie vor dem Kompilieren Ihres Codes sicher, dass Sie das Lambda-Paket von installiert haben. GitHub Dieses Modul stellt eine Implementierung der Laufzeitschnittstelle bereit, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet. Führen Sie den folgenden Befehl aus, um diese Bibliothek herunterzuladen.

```
go get github.com/aws/aws-lambda-go/lambda
```

Wenn Ihre Funktion das verwendet AWS SDK für Go, laden Sie den Standardsatz von SDK-Modulen zusammen mit allen AWS Service-API-Clients herunter, die für Ihre Anwendung erforderlich sind. Informationen zur Installation des SDK for Go finden Sie unter Erste Schritte mit der AWS SDK für Go V2.

Verwenden der bereitgestellten Laufzeitfamilie

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie eine OS-only Runtime (die `provided` Runtime-Familie), um Go Event Functions für Lambda bereitzustellen.

Um ein .zip-Bereitstellungspaket zu erstellen () macOS/Linux

1. Kompilieren Sie Ihre ausführbare Datei in dem Projektverzeichnis, das die Datei `main.go` Ihrer Anwendung enthält. Beachten Sie Folgendes:

   - Die ausführbare Datei muss `bootstrap` heißen. Weitere Informationen finden Sie unter Namenskonventionen für Handler.
   - Legen Sie Ihre Zielarchitektur für den Befehlssatz fest. OS-only Runtimes unterstützen sowohl arm64 als auch x86_64.
   - Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.

   arm64-Architektur:

   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   x86_64-Architektur:

   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (Optional) Möglicherweise müssen Sie Pakete mit Einstellung von `CGO_ENABLED=0` in Linux kompilieren:

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
    main.go
   ```

   Dieser Befehl erstellt ein stabiles Binärpaket für Standard-C-Bibliotheksversionen (`libc`), was auf Lambda und anderen Geräten unterschiedlich sein kann.

3. Erstellen Sie ein Bereitstellungspaket, indem Sie die ausführbare Datei in eine ZIP-Datei packen.

```
zip myFunction.zip bootstrap
```

> **Note**
>
> Fügen Sie die `bootstrap`-Datei dem Stamm der ZIP-Datei hinzu.

4. Erstellen der -Funktion Beachten Sie Folgendes:

   - Die Binärdatei muss `bootstrap` benannt werden, aber der Handlername kann beliebig sein. Weitere Informationen finden Sie unter Namenskonventionen für Handler.
   - Die `--architectures`-Option ist nur bei Verwendung von „arm64“ erforderlich. Der Standardwert ist „x86_64“.
   - Geben Sie für `--role` den Amazon-Ressourcennamen (ARN) der Ausführungsrolle an.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Erstellen einer ZIP-Datei unter Windows

Die folgenden Schritte zeigen, wie Sie das Tool build-lambda-zip für Windows von herunterladen, Ihre ausführbare Datei kompilieren und ein GitHub ZIP-Bereitstellungspaket erstellen.

> **Note**
>
> Falls noch nicht geschehen, müssen Sie git installieren und die ausführbare `git`-Datei Ihrer Windows- %PATH%-Umgebungsvariable hinzufügen.

Stellen Sie vor dem Kompilieren Ihres Codes sicher, dass Sie die Lambda-Bibliothek von installiert haben. GitHub Führen Sie den folgenden Befehl aus, um diese Bibliothek herunterzuladen.

```
go get github.com/aws/aws-lambda-go/lambda
```

Wenn Ihre Funktion das verwendet AWS SDK für Go, laden Sie den Standardsatz von SDK-Modulen zusammen mit allen AWS Service-API-Clients herunter, die für Ihre Anwendung erforderlich sind. Informationen zur Installation des SDK for Go finden Sie unter Erste Schritte mit der AWS SDK für Go V2.

Verwenden der bereitgestellten Laufzeitfamilie

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie eine OS-only Runtime (die `provided` Runtime-Familie), um Go Event Functions für Lambda bereitzustellen.

So erstellen Sie ein ZIP-Bereitstellungspaket (Windows)

1. Laden Sie das Tool build-lambda-zip von herunter. GitHub

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. Verwenden Sie das Tool aus Ihrer `GOPATH`, um eine ZIP-Datei zu erstellen. Bei einer Go-Standardinstallation befindet sich das Tool in der Regel unter `%USERPROFILE%\Go\bin`. Navigieren Sie andernfalls an den Installationsort der Go-Laufzeit und führen Sie einen der folgenden Schritte aus:

   cmd.exe

   Führen Sie in cmd.exe je nach Architektur des Zielbefehlssatzes einen der folgenden Schritte aus. OS-only Runtimes unterstützen sowohl arm64 als auch x86_64.

   Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.

   Example für die x86_64-Architektur

   ```
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

Example für die arm64-Architektur

```
set GOOS=linux
set GOARCH=arm64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

PowerShell

Führen Sie in PowerShell, abhängig von Ihrer Ziel-Befehlssatzarchitektur, einen der folgenden Befehle aus. OS-only Runtimes unterstützen sowohl arm64 als auch x86_64.

Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.

x86_64-Architektur:

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

arm64-Architektur:

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. Erstellen der -Funktion Beachten Sie Folgendes:

- Die Binärdatei muss `bootstrap` benannt werden, aber der Handlername kann beliebig sein. Weitere Informationen finden Sie unter Namenskonventionen für Handler.

- Die `--architectures`-Option ist nur bei Verwendung von „arm64“ erforderlich. Der Standardwert ist „x86_64“.
- Geben Sie für `--role` den Amazon-Ressourcennamen (ARN) der Ausführungsrolle an.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Go Lambda Event Functions mithilfe von ZIP-Dateien erstellen und aktualisieren

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Sie können Ihr .zip-Paket mithilfe der Lambda-Konsole, der und der AWS Command Line Interface Lambda-API bereitstellen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können eine vorhandene Container-Image-Funktion nicht konvertieren, um ein ZIP-Archiv zu verwenden. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Create Event Function aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie unter Runtime (Laufzeit) `provided.al2023` aus.
4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.

5. Wählen Sie „Event-Funktion erstellen“. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:
   a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.
   b. Klicken Sie auf Open.
   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Seite Event Functions der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Befehle Funktion erstellen und Funktionscode aktualisieren, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre ZIP-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip-Datei mit dem zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten erstellt. Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Sie können es auch verwenden AWS SAM , um eine Lambda-Funktion mithilfe eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion zu erstellen AWS SAM, können Sie Ihre ZIP-Datei in einem Amazon S3 S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer AWS SAM Vorlage spezifiziert die `AWS::Serverless::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri`- auf die Amazon S3 S3-URI, den Pfad zum lokalen Ordner oder FunctionCodeObjekt des Funktionscodes gesetzt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Wenn Ihre ZIP-Datei größer als 50 MB ist, müssen Sie sie nicht zuerst in einen Amazon S3 S3-Bucket hochladen. AWS SAM AWS SAM kann .zip-Pakete bis zur maximal zulässigen Größe von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei in finden Sie unter AWS::Serverless: AWS SAM:Function im Entwicklerhandbuch.AWS SAM

Beispiel: Verwenden AWS SAM um eine Go-Funktion mit der bereitgestellten.al2023 zu erstellen

1. Erstellen Sie eine AWS SAM Vorlage mit den folgenden Eigenschaften:

   - BuildMethod: Gibt den Compiler für Ihre Anwendung an. Verwenden Sie `go1.x`.
   - Runtime: Verwenden Sie `provided.al2023`.
   - CodeUri: Geben Sie den Pfad zu Ihrem Code ein.
   - Architectures: Verwenden Sie `[arm64]` für die arm64-Architektur. Verwenden Sie für die x86_64-Befehlssatzarchitektur die Option `[amd64]` oder entfernen Sie die Eigenschaft `Architectures`.

   Example template.yaml

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: 'AWS::Serverless-2016-10-31'
   Resources:
     HelloWorldFunction:
       Type: AWS::Serverless::Function
       Metadata:
         BuildMethod: go1.x
       Properties:
         CodeUri: hello-world/ # folder where your main program resides
         Handler: bootstrap
   ```

```
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. Verwenden Sie den Befehl sam build, um die ausführbare Datei zu kompilieren.

```
sam build
```

3. Verwenden Sie den Befehl sam deploy, um die Funktion in Lambda bereitzustellen.

```
sam deploy --guided
```

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit CloudFormation

Sie können verwenden CloudFormation , um eine Lambda-Funktion mithilfe eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die CloudFormation generierte ZIP-Datei darf 4 MB nicht überschreiten. Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei finden Sie unter AWS::Lambda: :Function im Benutzerhandbuch. CloudFormationCloudFormation

Stellen Sie Go Lambda Event Functions mit Container-Images bereit

Es gibt zwei Möglichkeiten, ein Container-Image für eine Go Lambda Event Function zu erstellen:

- Mit einem AWS OS-only Basisbild

  Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie ein OS-only Basis-Image, um Go-Images für Lambda zu erstellen. Um das Image mit Lambda kompatibel zu machen, müssen Sie das `aws-lambda-go/lambda`-Paket in das Image aufnehmen.
- Verwenden Sie ein nicht-AWS Basisbild

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie das `aws-lambda-go/lambda`-Paket in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

AWS Basis-Images für die Bereitstellung von Go-Funktionen

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie ein OS-only Basis-Image, um Go-Funktionen für Lambda bereitzustellen.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
OS-only Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029

Öffentliche Galerie von Amazon Elastic Container Registry: gallery.ecr. aws/lambda/bereitgestellt

Laufzeitschnittstellen-Clients von Go

Das `aws-lambda-go/lambda`-Paket enthält eine Implementierung der Laufzeitschnittstelle. Beispiele für die Verwendung von `aws-lambda-go/lambda` in Ihrem Image finden Sie unter Mit einem AWS OS-only Basisbild oder Verwenden Sie ein nicht-AWS Basisbild.

Mit einem AWS OS-only Basisbild

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie ein OS-only Basis-Image, um Container-Images für Go-Funktionen zu erstellen.

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
al2023	OS-only Laufzeit	Amazon Linux 2023	Dockerfile für Runtime auf OS-only GitHub	30. Juni 2029

Weitere Informationen zu diesen Basis-Images finden Sie in der öffentlichen Amazon-ECR-Galerie unter provided.

Sie müssen das go/lambdaaws-lambda-Paket in Ihren Go-Handler aufnehmen. Dieses Paket implementiert das Programmiermodell für Go, einschließlich der Laufzeitschnittstelle.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Go

Erstellen eines Images aus dem Basis-Image „provided.al2023“

Um eine Go-Funktion mit dem **mitgelieferten.al2023-Basisimage zu erstellen und bereitzustellen**

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir hello
cd hello
```

2. Initialisieren Sie ein neues Go-Modul.

```
go mod init example.com/hello-world
```

3. Fügen Sie die Lambda-Bibliothek als Abhängigkeit Ihres neuen Moduls hinzu.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Erstellen Sie eine Datei namens `main.go` und öffnen Sie diese dann in einem Text-Editor. Dies ist der Code für die Lambda-Funktion. Sie können den folgenden Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. Verwenden Sie einen Texteditor, um ein Dockerfile in Ihrem Projektverzeichnis zu erstellen.

   - Das folgende Beispiel für Dockerfile verwendet einen mehrstufigen Build auf der Docker-Dokumentationswebsite. Bei einem mehrstufigen Build können Sie in jedem Schritt ein anderes Basis-Image verwenden. Sie können ein Image verwenden, z. B. ein Go-Basisimage auf Docker Hub, um Ihren Code zu kompilieren und die ausführbare Binärdatei zu erstellen. Sie können dann ein anderes Image verwenden, z. B. `provided.al2023` in der `FROM`-Abschlusserklärung, um das Image zu definieren, das Sie für Lambda bereitstellen. Der mehrstufige Build trennt den Build-Prozess vom endgültigen Deployment-Image, sodass das endgültige Image nur die Dateien enthält, die zur Ausführung der Anwendung benötigt werden.
   - Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.
   - Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   Example— Multi-stage Dockerfile erstellen

   > **Note**
   >
   > Stellen Sie sicher, dass die Version von Go, die Sie in der Docker-Datei angeben (z. B. `golang:1.20`), dieselbe Version von Go ist, mit der Sie Ihre Anwendung erstellt haben.

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
```

```
ENTRYPOINT [ "./main" ]
```

6. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um Ihr Image lokal zu testen. Der Laufzeit-Schnittstellen-Emulator ist im `provided.al2023`-Basis-Image enthalten.

So führen Sie den Laufzeit-Schnittstellen-Emulator auf Ihrem lokalen Computer aus

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `./main` ist ENTRYPOINT aus Ihrem Dockerfile.

```
docker run -d -p 9000:8080 \
--entrypoint /usr/local/bin/aws-lambda-rie \
docker-image:test ./main
```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis mit einem curl-Befehl an den folgenden Endpunkt:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Für einige Funktionen ist möglicherweise eine JSON-Nutzlast erforderlich. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden Sie ein nicht-AWS Basisbild

Sie können ein Container-Image für Go aus einem AWS Nicht-Base-Image erstellen. Das Beispiel-Dockerfile in den folgenden Schritten verwendet ein Alpine-Basis-Image.

Sie müssen das go/lambdaaws-lambda-Paket in Ihren Go-Handler aufnehmen. Dieses Paket implementiert das Programmiermodell für Go, einschließlich der Laufzeitschnittstelle.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Go

Erstellen eines Images aus einem alternativen Basis-Image

Erstellen und Bereitstellen einer Go-Funktion mit einem Alpine-Basis-Image

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir hello
cd hello
```

2. Initialisieren Sie ein neues Go-Modul.

```
go mod init example.com/hello-world
```

3. Fügen Sie die Lambda-Bibliothek als Abhängigkeit Ihres neuen Moduls hinzu.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Erstellen Sie eine Datei namens `main.go` und öffnen Sie diese dann in einem Text-Editor. Dies ist der Code für die Lambda-Funktion. Sie können den folgenden Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. Verwenden Sie einen Texteditor, um ein Dockerfile in Ihrem Projektverzeichnis zu erstellen. Das folgende Beispiel-Dockerfile verwendet ein Alpine-Basis-Image. Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem

standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

Example Dockerfile

> **Note**
>
> Stellen Sie sicher, dass die Version von Go, die Sie in der Docker-Datei angeben (z. B. `golang:1.20`), dieselbe Version von Go ist, mit der Sie Ihre Anwendung erstellt haben.

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build
COPY main.go .
RUN go build -o main main.go
# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

6. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `/main` ist `ENTRYPOINT` aus Ihrem Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /main
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /main
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

   Wenn erfolgreich, sehen Sie eine Antwort wie diese:

   ```
   {
       "repository": {
           "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
   world",
           "registryId": "111122223333",
           "repositoryName": "hello-world",
           "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world",
           "createdAt": "2023-03-09T10:39:01+00:00",
           "imageTagMutability": "MUTABLE",
           "imageScanningConfiguration": {
               "scanOnPush": true
   ```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

> **Note**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Go Lambda Event Functions

Wir empfehlen nicht, Ebenen zu verwenden, um Abhängigkeiten für in Go geschriebene Lambda-Funktionen zu verwalten. Dies liegt daran, dass Lambda-Funktionen in Go zu einer einzigen ausführbaren Datei kompiliert werden, die Sie Lambda bei der Bereitstellung Ihrer Funktion zur Verfügung stellen. Diese ausführbare Datei enthält Ihren kompilierten Funktionscode zusammen mit all seinen Abhängigkeiten. Die Verwendung von Ebenen verkompliziert nicht nur diesen Vorgang, sondern führt auch zu längeren Kaltstartzeiten, da Ihre Funktionen während der Init-Phase zusätzliche Assemblys manuell in den Speicher laden müssen.

Um externe Abhängigkeiten mit Ihren Go-Handlern zu verwenden, nehmen Sie sie direkt in Ihr Bereitstellungspaket auf. Auf diese Weise vereinfachen Sie den Bereitstellungsprozess und nutzen auch die Vorteile der integrierten Go-Compiler-Optimierungen. Ein Beispiel dafür, wie Sie eine Abhängigkeit wie das AWS SDK for Go in Ihrer Funktion importieren und verwenden, finden Sie unter the section called “Handler”.

Go Lambda Event Functions protokollieren und überwachen

AWS Lambda überwacht Lambda Event Functions automatisch in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Funktion umfasst eine CloudWatch Logs-Protokollgruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Lambda Event Function-Logs an CloudWatch Logs senden.

Auf dieser Seite wird beschrieben, wie Sie eine Protokollausgabe aus dem Code Ihrer Lambda-Funktion erstellen und mithilfe der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie Methoden für das Fmt-Paket verwenden oder eine Protokollierungsbibliothek, die zu `stdout` oder `stderr` schreibt. Im folgenden Beispiel wird das Protokollpaket verwendet.

Example main.go — Protokollierung

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
  }
```

Example Protokollformat

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 243 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

Die Go-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.

- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Go-Code instrumentieren AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können es verwenden X-Ray, um eine Anfrage zu verfolgen, während sie Ressourcen in Ihrer Anwendung durchquert, zu denen Lambda-Funktionen und andere AWS Dienste gehören können.

Um Ablaufverfolgungsdaten an zu senden X-Ray, können Sie eine von zwei SDK-Bibliotheken verwenden:

- AWS Distribution for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des (OTel) SDK. OpenTelemetry
- AWS X-Ray SDK for Go — Ein SDK zum Generieren und Senden von Trace-Daten an X-Ray.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray Dienst zu senden. Anschließend können Sie X-Ray die Leistungskennzahlen Ihrer Anwendung anzeigen, filtern und Einblicke in sie gewinnen, um Probleme und Optimierungsmöglichkeiten zu identifizieren.

> **Important**
>
> Die SDKs X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von angeboten wird. AWS Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgängige Ablaufverfolgung implementieren. X-Ray Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Auswahl zwischen der AWS Distribution for Open Telemetry und SDKs. X-Ray

Sections

- Verwenden von ADOT zur Instrumentierung Ihrer Go-Funktionen
- Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Go-Funktionen
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Die Ablaufverfolgung wird aktiviert mit CloudFormation
- Interpretieren und X-Ray Trace

Verwenden von ADOT zur Instrumentierung Ihrer Go-Funktionen

ADOT bietet vollständig verwaltete Lambda-Ebenen, die alles packen, was Sie zum Sammeln von Telemetriedaten mit dem OTel-SDK benötigen. Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Ebenen auch für die benutzerdefinierte Initialisierung von OTel konfigurieren. Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Go-Laufzeiten können Sie den AWS -verwaltete Lambda-Ebene für ADOT Go hinzufügen, um Ihre Funktionen automatisch zu instrumentieren. Eine ausführliche Anleitung zum Hinzufügen dieser Ebene finden Sie unter AWS Distro for OpenTelemetry Lambda Support for Go in der ADOT-Dokumentation. Anweisungen zur Konvertierung von Lambda-Telemetrie-API-Ereignissen in OpenTelemetry Spans finden Sie unter. the section called “Konvertieren von Ereignissen in OTel-Spans”

Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer Go-Funktionen

Um Details zu Aufrufen aufzuzeichnen, die Ihre Lambda-Funktion an andere Ressourcen in Ihrer Anwendung tätigt, können Sie auch das AWS X-Ray SDK for Go verwenden. Um das SDK zu erhalten, laden Sie das SDK aus seinem GitHub Repository herunter mit`go get`:

```
go get github.com/aws/aws-xray-sdk-go
```

Um AWS SDK-Clients zu instrumentieren, übergeben Sie den Client an die `xray.AWS()` Methode. Anschließend können Sie Aufrufe verfolgen, indem Sie die `WithContext`-Version der Methode verwenden.

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

Nachdem Sie die richtigen Abhängigkeiten hinzugefügt und die erforderlichen Codeänderungen vorgenommen haben, aktivieren Sie die Ablaufverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpretieren und X-Ray Trace

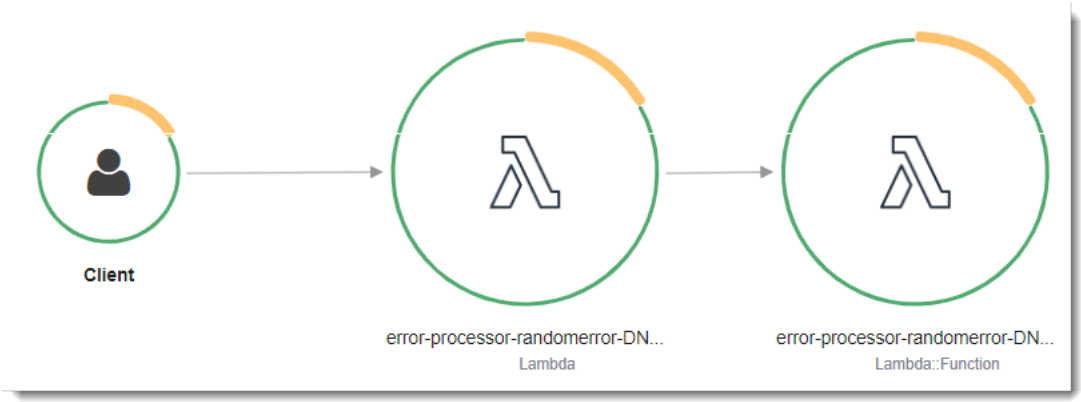
Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen X-Ray. Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray Dienstdiagramm zeigt Informationen zu Ihrer Anwendung und all ihren Komponenten. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK X-Ray, um Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.



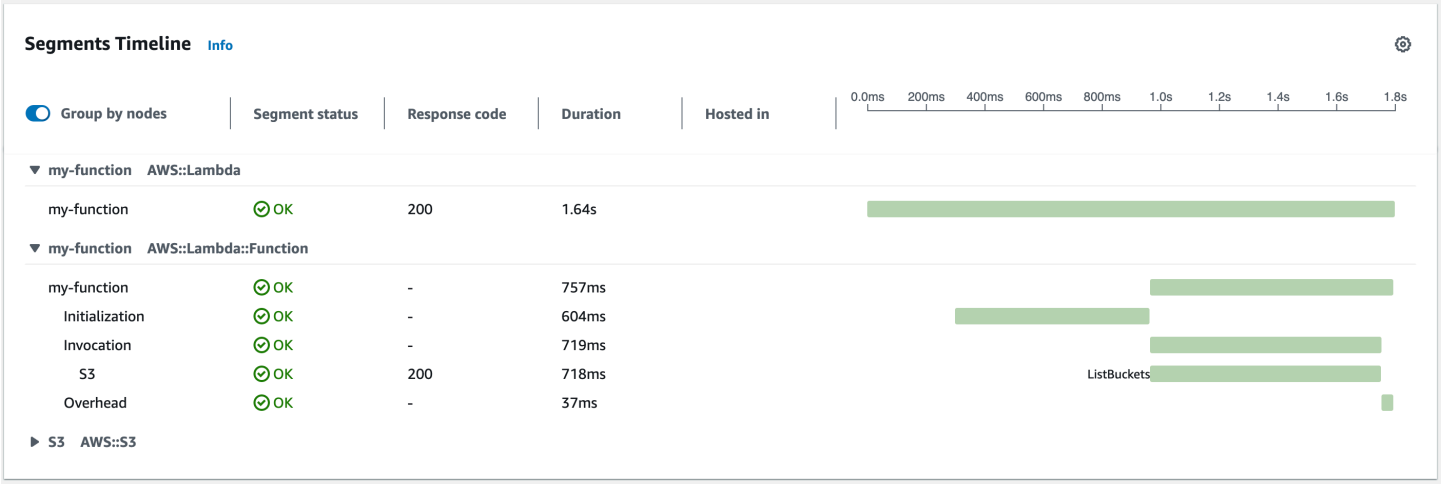
X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

In X-Ray zeichnet ein Trace Informationen über eine Anfrage auf, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem.

Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können Sie geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollnachrichten und Trace-Segmenten feststellen, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.

- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray -SDK für Go im AWS X-Ray -Entwicklerhandbuch.

> **Preisgestaltung**
>
> Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit kostenlos nutzen. AWS Bei Überschreitung dieses Schwellenwerts X-Ray fallen Gebühren für die Speicherung und den Abruf von Trace an. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Lambda-Ereignisfunktionen mit C# erstellen

Sie können Ihre .NET-Anwendung als Lambda-Ereignisfunktion mit der verwalteten .NET 8-Laufzeit, einer benutzerdefinierten Laufzeit oder einem Container-Image ausführen. Nachdem Ihr Anwendungscode kompiliert wurde, können Sie ihn entweder als ZIP-Datei oder als Container-Image für Lambda bereitstellen. Lambda stellt die folgenden Laufzeiten für .NET-Sprachen bereit:

Informationen zur Verwendung von.NET-Layern finden Sie unterthe section called “Layer”. Hinweise zum Testen Ihrer C#-Funktionen finden Sie unterthe section called “Testen”.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 10	`dotnet10`	Amazon Linux 2023	14. November 2028	14. Dezember 2028	15. Januar 2029

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 9 (nur Container)	`dotnet9`	Amazon Linux 2023	10. November 2026	Nicht geplant	Nicht geplant
.NET 8	`dotnet8`	Amazon Linux 2023	10. November 2026	29. Juli 2027	31. August 2027

Einrichten der .NET-Entwicklungsumgebung

Um Ihre Lambda-Ereignisfunktionen zu entwickeln und zu erstellen, können Sie jede der allgemein verfügbaren.NET-Integrated Development Environments (IDEs) verwenden, einschließlich Microsoft Visual Studio, Visual Studio Code und JetBrains Rider. Zur Vereinfachung Ihrer Entwicklungserfahrung AWS bietet es eine Reihe von .NET-Projektvorlagen sowie die `Amazon.Lambda.Tools` Befehlszeilenschnittstelle (CLI).

Führen Sie die folgenden .NET-CLI-Befehle aus, um diese Projektvorlagen und Befehlszeilentools zu installieren.

Installieren der .NET-Projektvorlagen

Zur Installation der Projektvorlagen führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

Installation und Aktualisierung der CLI-Tools

Führen Sie die folgenden Befehle aus, um die `Amazon.Lambda.Tools`-CLI zu installieren, zu aktualisieren und zu deinstallieren.

Installieren der Befehlszeilen-Tools:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Aktualisieren der Befehlszeilen-Tools:

```
dotnet tool update -g Amazon.Lambda.Tools
```

Deinstallieren der Befehlszeilen-Tools:

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

Definieren Sie den Lambda-Ereignisfunktionshandler in C#

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Ereignisfunktionshandlern in C# arbeiten, um mit der.NET-verwalteten Runtime zu arbeiten, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine C#-Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdatei als Beleg erstellt und diese Datei in einen Bucket in Amazon Simple Storage Service (S3) stellt. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellungspaket” oder the section called “Bereitstellen von Container-Images”.

Themen

- Einrichten Ihres C#-Handler-Projekts
- Beispiel für einen C#-Lambda-Ereignisfunktionscode
- Handler für Klassenbibliotheken
- Ausführbare Assembly-Handler
- Gültige Handler-Signaturen für C#-Funktionen
- Namenskonventionen für Handler
- Serialisierung in C#-Lambda-Ereignisfunktionen
- File-based Funktionen
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Verwendung der SDK für .NET v3 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Vereinfachen Sie den Funktionscode mit dem Lambda Annotations Framework
- Bewährte Codemethoden für Lambda-Ereignisfunktionen in C#

Einrichten Ihres C#-Handler-Projekts

Wenn Sie mit Lambda-Ereignisfunktionen in C# arbeiten, umfasst der Prozess das Schreiben Ihres Codes und das anschließende Bereitstellen Ihres Codes in Lambda. Es gibt zwei

verschiedene Ausführungsmodelle für die Bereitstellung von Lambda-Funktionen in .NET: den Klassenbibliotheksansatz und den Ansatz für ausführbare Assemblys.

Beim Klassenbibliotheksansatz packen Sie Ihren Funktionscode als .NET-Assembly (`.dll`) und stellen ihn mit der verwalteten .NET-Laufzeit (`dotnet8`) in Lambda bereit. Für den Namen des Handlers erwartet Lambda eine Zeichenfolge im Format `AssemblyName::Namespace.Classname::Methodname`. Während der Initialisierungsphase der Funktion wird die Klasse Ihrer Funktion initialisiert und der gesamte Code im Konstruktor wird ausgeführt.

Beim Ansatz für ausführbare Assemblys verwenden Sie das Feature für Anweisungen der obersten Ebene, die erstmals in C# 9 eingeführt wurde. Dieser Ansatz generiert eine ausführbare Assembly, die Lambda immer dann ausführt, wenn sie einen Aufrufbefehl für Ihre Funktion empfängt. Bei diesem Ansatz verwenden Sie außerdem die verwaltete .NET-Laufzeit (`dotnet8`). Für den Namen des Handlers geben Sie in Lambda den Namen der ausführbaren Assembly an, die ausgeführt werden soll.

Das Hauptbeispiel auf dieser Seite veranschaulicht den Klassenbibliotheksansatz. Sie können Ihr C#-Lambda-Projekt auf verschiedene Arten initialisieren. Am einfachsten ist es jedoch, die .NET-CLI mit der `Amazon.Lambda.Tools`-CLI zu verwenden. Richten Sie die `Amazon.Lambda.Tools`-CLI ein, indem Sie die Schritte unter the section called “Entwicklungsumgebung” befolgen. Initialisieren Sie das Projekt dann mit dem folgenden Befehl:

```
dotnet new lambda.EmptyFunction --name ExampleCS
```

Dieser Befehl generiert die folgende Dateistruktur:

```
/project-root
    # src
        # ExampleCS
            # Function.cs (contains main handler)
            # Readme.md
            # aws-lambda-tools-defaults.json
            # ExampleCS.csproj
    # test
         # ExampleCS.Tests
            # FunctionTest.cs (contains main handler)
            # ExampleCS.Tests.csproj
```

In dieser Dateistruktur befindet sich die Hauptlogik des Handlers für Ihre Funktion in der Datei `Function.cs`.

Beispiel für einen C#-Lambda-Ereignisfunktionscode

Das folgende Beispiel für einen C#-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdatei als Beleg und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`Function.cs`**Lambda-Ereignisfunktion

```
using System;
using System.Text;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;

// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

public class OrderHandler
{
    private static readonly AmazonS3Client s3Client = new();

    public async Task<string> HandleRequest(Order order, ILambdaContext context)
    {
        try
        {
            string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
            if (string.IsNullOrWhiteSpace(bucketName))
            {
                throw new ArgumentException("RECEIPT_BUCKET environment variable is not
 set");
```

```
            }

            string receiptContent = $"OrderID: {order.OrderId}\nAmount:
 ${order.Amount:F2}\nItem: {order.Item}";
            string key = $"receipts/{order.OrderId}.txt";

            await UploadReceiptToS3(bucketName, key, receiptContent);

            context.Logger.LogInformation($"Successfully processed order
 {order.OrderId} and stored receipt in S3 bucket {bucketName}");
            return "Success";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to process order: {ex.Message}");
            throw;
        }
    }

    private async Task UploadReceiptToS3(string bucketName, string key, string
 receiptContent)
    {
        try
        {
            var putRequest = new PutObjectRequest
            {
                BucketName = bucketName,
                Key = key,
                ContentBody = receiptContent,
                ContentType = "text/plain"
            };

            await s3Client.PutObjectAsync(putRequest);
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to upload receipt to S3: {ex.Message}", ex);
        }
    }
}
```

Diese `Function.cs`-Datei enthält die folgenden Abschnitte des Codes:

- `using`-Anweisungen: Damit können Sie C#-Klassen importieren, die für Ihre Lambda-Funktion erforderlich sind.
- `[assembly: LambdaSerializer(...)]`: `LambdaSerializer` ist ein Assembly-Attribut, das Lambda anweist, JSON-Ereignisnutzdaten automatisch in C#-Objekte zu konvertieren, bevor sie an Ihre Funktion übergeben werden.
- `namespace ExampleLambda`: Dies definiert den Namespace. In C# muss der Namespace-Name nicht mit dem Dateinamen übereinstimmen.
- `public class Order {...}`: Hiermit wird die Form des erwarteten Eingabeereignisses definiert.
- `public class OrderHandler {...}`: Hiermit wird die C#-Klasse definiert. Darin definieren Sie die Haupt-Handler-Methode und alle anderen Hilfsmethoden.
- `private static readonly AmazonS3Client s3Client = new();`: Dadurch wird ein Amazon-S3-Client mit der standardmäßigen Anbieterkette von Anmeldeinformationen außerhalb der Haupt-Handler-Methode initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen.
- `public async ... HandleRequest (Order order, ILambdaContext context)`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `private async Task UploadReceiptToS3(...) {}`: Dies ist eine Hilfsmethode, auf die von der `handleRequest`-Haupthandler-Methode verwiesen wird.

Da für diese Funktion ein SDK-Client für Amazon S3 erforderlich ist, müssen Sie sie zu den Abhängigkeiten Ihres Projekts hinzufügen. Dazu können Sie zu `src/ExampleCS` navigieren und den folgenden Befehl ausführen:

```
dotnet add package AWSSDK.S3
```

Hinzufügen von Metadateninformationen zu „aws-lambda-tools-defaults.json“

Standardmäßig enthält die generierte Datei `aws-lambda-tools-defaults.json` keine `profile`- oder `region`-Informationen für Ihre Funktion. Aktualisieren Sie außerdem die `function-handler`-Zeichenfolge auf den richtigen Wert (`ExampleCS::ExampleLambda.OrderHandler::HandleRequest`). Sie können diese Aktualisierung manuell vornehmen und die erforderlichen Metadaten hinzufügen, um ein bestimmtes Anmeldeinformationsprofil und eine bestimmte Region für Ihre Funktion zu verwenden. Die `aws-lambda-tools-defaults.json`-Datei sollte etwa wie folgt aussehen:

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual Studio
 and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "default",
  "region": "us-east-1",
  "configuration": "Release",
  "function-architecture": "x86_64",
  "function-runtime": "dotnet8",
  "function-memory-size": 512,
  "function-timeout": 30,
  "function-handler": "ExampleCS::ExampleLambda.OrderHandler::HandleRequest"
}
```

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

Handler für Klassenbibliotheken

Der Hauptbeispielcode auf dieser Seite veranschaulicht den Klassenbibliotheks-Handler. Klassenbibliotheks-Handler haben die folgende Struktur:

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace NAMESPACE;

...

public class CLASSNAME {
    public async Task<string> METHODNAME (...) {
    ...
    }
}
```

Wenn Sie eine Lambda-Ereignisfunktion erstellen, müssen Sie Lambda Informationen über den Handler Ihrer Funktion in Form einer Zeichenfolge im Feld Handler zur Verfügung stellen. Dadurch wird Lambda mitgeteilt, welche Methode in Ihrem Code ausgeführt werden soll, wenn Ihre Funktion aufgerufen wird. In C# lautet das Format der Handler-Zeichenfolge für Klassenbibliotheks-Handler `ASSEMBLY::TYPE::METHOD`. Dabei gilt Folgendes:

- `ASSEMBLY` ist der Name der .NET-Assembly-Datei für Ihre Anwendung. Wenn Sie die `Amazon.Lambda.Tools`-CLI zur Erstellung Ihrer Anwendung verwenden und den Assembly-Namen nicht über die Eigenschaft `AssemblyName` in der `.csproj`-Datei festlegen, ist `ASSEMBLY` einfach der Name Ihrer `.csproj`-Datei.
- `TYPE` ist der vollständige Name des Handler-Typs, der `NAMESPACE.CLASSNAME` lautet.
- `METHOD` ist der Name der Haupt-Handler-Methode in Ihrem Code. Er lautet `METHODNAME`.

Wenn die Assembly im Hauptbeispielcode auf dieser Seite den Namen `ExampleCS` trägt, lautet die vollständige Zeichenfolge für den Handler `ExampleCS::ExampleLambda.OrderHandler::HandleRequest`.

Ausführbare Assembly-Handler

Sie können Lambda-Funktionen in C# auch als ausführbare Assembly definieren. Ausführbare Assembly-Handler verwenden die Funktion für Anweisungen der obersten Ebene von C#, bei der der Compiler die `Main()` Methode generiert und Ihren Funktionscode darin ablegt. Bei der Verwendung ausführbarer Assemblys muss die Lambda-Laufzeit gebootet werden.

Verwenden Sie dazu die `LambdaBootstrapBuilder.Create`-Methode in Ihrem Code. Die Eingaben für diese Methode stellen die Haupt-Handler-Funktion sowie den Lambda-Serialisierer dar, die verwendet werden sollen. Im Anschluss sehen Sie ein Beispiel für einen Handler für ausführbare Assemblys in C#:

```
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();
```

```
async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
{
    var id = apigProxyEvent.PathParameters["id"];
    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
};
```

Im Feld Handler für Handler für ausführbare Assemblys ist die Handler-Zeichenfolge, die Lambda mitteilt, wie Ihr Code ausgeführt werden soll, der Name der Assembly. In diesem Beispiel ist dies `GetProductHandler`.

Gültige Handler-Signaturen für C#-Funktionen

In C# können gültige Lambda-Handler-Signaturen zwischen 0 und 2 Argumente enthalten. Normalerweise hat Ihre Handler-Signatur zwei Argumente, wie im Hauptbeispiel gezeigt:

```
public async Task<string> HandleRequest(Order order, ILambdaContext context)
```

Wenn Sie zwei Argumente angeben, muss das erste Argument die Ereigniseingabe und das zweite Argument das Lambda-Kontextobjekt sein. Beide Argumente sind optional. Die folgenden sind beispielsweise ebenfalls gültige Lambda-Handler-Signaturen in C#:

- `public async Task<string> HandleRequest()`
- `public async Task<string> HandleRequest(Order order)`
- `public async Task<string> HandleRequest(ILambdaContext context)`

Neben der Basissyntax der Handler-Signatur gibt es einige zusätzliche Einschränkungen:

- Sie können das Schlüsselwort `unsafe` nicht in der Handler-Signatur verwenden. Sie können jedoch den `unsafe`-Kontext innerhalb der Handler-Methode und ihrer Abhängigkeiten verwenden. Weitere Informationen finden Sie unter unsafe (C#-Referenz) auf der Microsoft-Dokumentationswebsite.

- Der Handler kann das `params` Schlüsselwort nicht verwenden oder es `ArgIterator` als Eingabe- oder Rückgabeparameter verwenden. Diese Schlüsselwörter unterstützen eine variable Anzahl von Parametern. Ihr Handler kann maximal zwei Argumente akzeptieren.
- Der Handler kann keine generische Methode sein. Mit anderen Worten: Er kann keine generischen Typparameter wie <T> verwenden.
- Lambda unterstützt keine asynchronen Handler mit `async void` in der Signatur.

Namenskonventionen für Handler

Lambda-Handler in C# haben keine strengen Benennungsbeschränkungen. Sie müssen jedoch sicherstellen, dass Sie Lambda bei der Bereitstellung Ihrer Funktion die richtige Handler-Zeichenfolge zur Verfügung stellen. Die richtige Handler-Zeichenfolge hängt davon ab, ob Sie einen Klassenbibliotheks-Handler oder einen Handler für ausführbare Assemblys bereitstellen.

Sie können zwar einen beliebigen Namen für Ihren Handler verwenden, Funktionsnamen in C# sind jedoch generell in PascalCase. Auch wenn der Dateiname nicht mit dem Klassennamen oder dem Handler-Namen übereinstimmen muss, ist es im Allgemeinen eine bewährte Methode, einen Dateinamen wie `OrderHandler.cs` zu verwenden, wenn der Klassenname `OrderHandler` lautet. Sie können den Dateinamen in diesem Beispiel beispielsweise von `Function.cs` in `OrderHandler.cs` ändern.

Serialisierung in C#-Lambda-Ereignisfunktionen

JSON ist das gängigste und standardmäßigste Eingabeformat für Lambda Event Functions. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

In C# können Sie die Form des erwarteten Eingabeereignisses in einer Klasse definieren. In diesem Beispiel definieren wir die `Order`-Klasse, um diese Eingabe zu modellieren:

```
public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
```

```
    public string Item { get; set; } = string.Empty;
}
```

Wenn Ihre Lambda-Funktion andere Eingabe- oder Ausgabetypen als ein `Stream`-Objekt verwendet, müssen Sie eine Serialisierungsbibliothek zu Ihrer Anwendung hinzufügen. Anschließend können Sie die JSON-Eingabe in eine Instanz der von Ihnen definierten Klasse konvertieren. Es gibt zwei Methoden der Serialisierung für C#-Funktionen in Lambda: reflexionsbasierte Serialisierung und quellgenerierte Serialisierung.

Reflection-based Serialisierung

AWS stellt vorgefertigte Bibliotheken bereit, die Sie schnell zu Ihrer Anwendung hinzufügen können. Diese Bibliotheken implementieren Serialisierung mithilfe von Reflexion. Verwenden Sie eins der folgenden Pakete, um die reflexionsbasierte Serialisierung zu implementieren:

- `Amazon.Lambda.Serialization.SystemTextJson`: Im Backend nutzt dieses Paket `System.Text.Json` zur Ausführung von Serialisierungsaufgaben.
- `Amazon.Lambda.Serialization.Json`: Im Backend nutzt dieses Paket `Newtonsoft.Json` zur Ausführung von Serialisierungsaufgaben.

Sie können auch Ihre eigene Serialisierungsbibliothek erstellen, indem Sie die `ILambdaSerializer`-Schnittstelle implementieren; diese ist als Teil der `Amazon.Lambda.Core`-Bibliothek verfügbar. Diese Schnittstelle definiert zwei Methoden:

- `T Deserialize<T>(Stream requestStream);`

  Sie implementieren diese Methode für die Deserialisierung der Anfragenutzlast von der `Invoke`-API in das Objekt, das Ihrem Lambda-Funktions-Handler übergeben wird.
- `T Serialize<T>(T response, Stream responseStream);`

  Sie implementieren diese Methode, um das von Ihrem Lambda-Funktions-Handler zurückgegebene Ergebnis in die Antwort-Nutzlast zu serialisieren, die der `Invoke`-API-Vorgang zurückgibt.

Das Hauptbeispiel auf dieser Seite verwendet reflektionsbasierte Serialisierung. Reflection-based Die Serialisierung ist sofort einsatzbereit AWS Lambda und erfordert keine zusätzliche Einrichtung, weshalb sie aus Gründen der Einfachheit eine gute Wahl ist. Es ist jedoch mehr Funktionsspeicher erforderlich. Aufgrund von Laufzeitreflexionen kann es auch zu höheren Funktionslatenzen kommen.

Source-generated Serialisierung

Bei der quellgenerierten Serialisierung wird der Serialisierungscode zur Kompilierzeit generiert. Dies macht Reflexion überflüssig und kann die Leistung Ihrer Funktion verbessern. Gehen Sie wie folgt vor, um die quellgenerierte Serialisierung in Ihrer Funktion zu verwenden:

- Erstellen Sie eine neue Teilklasse, die von `JsonSerializerContext` erbt, und fügen Sie `JsonSerializable`-Attribute für alle Typen hinzu, die serialisiert oder deserialisiert werden müssen.
- Konfigurieren Sie das `LambdaSerializer` so, dass es ein `SourceGeneratorLambdaJsonSerializer<T>` verwendet.
- Aktualisieren Sie alle manuellen Serialisierungen und Deserialisierungen in Ihrem Anwendungscode, um die neu erstellte Klasse zu verwenden.

Das folgende Beispiel zeigt, wie Sie das Hauptbeispiel auf dieser Seite, das reflexionsbasierte Serialisierung verwendet, so ändern können, dass stattdessen die quellgenerierte Serialisierung verwendet wird.

```
using System.Text.Json;
using System.Text.Json.Serialization;

...

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

[JsonSerializable(typeof(Order))]
public partial class OrderJsonContext : JsonSerializerContext {}

public class OrderHandler
{

    ...

    public async Task<string> HandleRequest(string input, ILambdaContext context)
    {
```

```
    var order = JsonSerializer.Deserialize(input, OrderJsonContext.Default.Order);

    ...

    }

}
```

Source-generated Die Serialisierung erfordert mehr Einrichtung als die reflexionsbasierte Serialisierung. Funktionen, die quellgenerierte Serialisierung verwenden, verbrauchen jedoch tendenziell weniger Speicher und weisen aufgrund der Codegenerierung während der Kompilierung eine bessere Leistung auf. Um Kaltstarts von Funktionen zu vermeiden, sollten Sie in Erwägung ziehen, zur quellgenerierten Serialisierung zu wechseln.

> **Note**
>
> Wenn Sie die native Ahead-of-Time-Kompilierung (AOT) mit Lambda verwenden möchten, müssen Sie die quellgenerierte Serialisierung verwenden.

File-based Funktionen

Mit dateibasierten Apps, die in .NET 10 eingeführt wurden, können Sie.NET-Anwendungen aus einer einzigen `.cs` Datei ohne Datei- oder `.csproj` Verzeichnisstruktur erstellen. Lambda unterstützt dateibasierte Funktionen, beginnend mit.NET 10. Sie bieten eine optimierte, einfache Möglichkeit, Lambda-Ereignisfunktionen in C# zu erstellen.

Der schnellste Weg, um mit der Erstellung einer auf C#-Dateien basierenden Lambda-Funktion zu beginnen, ist die Verwendung des Pakets. `Amazon.Lambda.Templates` Zur Installation dieses Pakets führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

Erstellen Sie als Nächstes eine auf einer C#-Datei basierende Lambda-Beispielfunktion:

```
dotnet new lambda.FileBased -n MyLambdaFunction
```

File-based Funktionen verwenden ausführbare Assembly-Handler. Sie müssen daher das `Amazon.Lambda.RuntimeSupport` NuGet Paket einschließen und die

`LambdaBootstrapBuilder.Create` Methode verwenden, um die .NET-Handlerfunktion für den Ereignistyp zu registrieren und den.NET Lambda-Runtime-Client zu starten.

File-based Funktionen verwenden standardmäßig .NET Native AOT, was eine quellgenerierte Serialisierung erfordert. Sie können Native AOT deaktivieren, indem Sie es in Ihrer Quelldatei angeben`#:property PublishAot=false`. Weitere Informationen zur Verwendung von Native AOT in Lambda finden Sie unter. the section called “Native AOT-Kompilierung”

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In diesem Beispiel ist das Kontextobjekt vom Typ `Amazon.Lambda.Core.ILambdaContext` und das zweite Argument der Haupthandlerfunktion.

```
public async Task<string> HandleRequest(Order order, ILambdaContext context) {
    ...
}
```

Das Kontextobjekt ist eine optionale Eingabe. Weitere Informationen zu gültigen akzeptierten Handlersignaturen finden Sie unter the section called “Gültige Handler-Signaturen für C#-Funktionen”.

Das Kontextobjekt ist nützlich, um Funktionsprotokolle für Amazon zu erstellen CloudWatch. Sie können die `context.getLogger()`-Methode verwenden, um ein `LambdaLogger`-Objekt für die Protokollierung abzurufen. In diesem Beispiel können wir den Logger verwenden, um eine Fehlermeldung zu protokollieren, falls bei der Verarbeitung aus irgendeinem Grund ein Fehler auftritt:

```
context.Logger.LogError($"Failed to process order: {ex.Message}");
```

Außerhalb der Protokollierung können Sie das Kontextobjekt auch für die Funktionsüberwachung verwenden. Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Kontext”.

Verwendung der SDK für .NET v3 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von SDK für .NET v3.

> **Note**
>
> Die SDK für .NET (v2) ist veraltet. Wir empfehlen, nur Version 3 zu verwenden. SDK für .NET

Mit dem folgenden `Amazon.Lambda.Tools`-Befehl können Sie Ihrem Projekt SDK-Abhängigkeiten hinzufügen:

```
dotnet add package <package_name>
```

Im Hauptbeispiel auf dieser Seite müssen wir beispielsweise die Amazon-S3-API verwenden, um einen Beleg in S3 hochzuladen. Wir können den SDK-Client für Amazon S3 mit dem folgenden Befehl importieren:

```
dotnet add package AWSSDK.S3
```

Dieser Befehl fügt die Abhängigkeit zu Ihrem Projekt hinzu. In der `.csproj`-Datei Ihres Projekts sollte außerdem eine Zeile ähnlich der folgenden angezeigt werden:

```
<PackageReference Include="AWSSDK.S3" Version="3.7.2.18" />
```

Importieren Sie dann die Abhängigkeiten direkt in Ihren C#-Code:

```
using Amazon.S3;
using Amazon.S3.Model;
```

Der Beispielcode initialisiert dann wie folgt einen Amazon-S3-Client (unter Verwendung der standardmäßigen Anbieterkette für Anmeldeinformationen):

```
private static readonly AmazonS3Client s3Client = new();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS Diensten verwenden. Der Beispielcode ruft die Amazon S3 `PutObject`-API wie folgt auf:

```
var putRequest = new PutObjectRequest
{
```

```
    BucketName = bucketName,
    Key = key,
    ContentBody = receiptContent,
    ContentType = "text/plain"
};

await s3Client.PutObjectAsync(putRequest);
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `System.Environment.GetEnvironmentVariable`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit den folgenden Codezeilen auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
if (string.IsNullOrWhiteSpace(bucketName))
{
    throw new ArgumentException("RECEIPT_BUCKET environment variable is not set");
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code und den Klassenkonstruktor während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb der Haupt-Handler-Methode. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis verarbeitet, was zu längeren Verarbeitungszeiten führen kann. Nachfolgende Ereignisse sind viel schneller, da Lambda den Client nicht erneut initialisieren muss.

Vereinfachen Sie den Funktionscode mit dem Lambda Annotations Framework

Lambda Annotations ist ein Framework für .NET 8, das das Schreiben von Lambda-Funktionen mit C# vereinfacht. Das Annotations-Framework verwendet Quellgeneratoren, um Code zu generieren, der vom Lambda-Programmiermodell in den vereinfachten Code übersetzt wird. Mit dem Annotations-Framework können Sie einen Großteil des Codes in einer Lambda-Funktion ersetzen, die mit dem regulären Programmiermodell geschrieben wurde. Code, der mit dem Framework geschrieben wurde,

verwendet einfachere Ausdrücke, sodass Sie sich auf Ihre Geschäftslogik konzentrieren können. Beispiele finden Sie Amazon.Lambda.Annotationsin der Nuget-Dokumentation.

Ein Beispiel für eine vollständige Anwendung, die Lambda-Anmerkungen verwendet, finden Sie im PhotoAssetManagerBeispiel im `awsdocs/aws-doc-sdk-examples` GitHub Repository. Die Hauptdatei `Function.cs` im Verzeichnis `PamApiAnnotations` verwendet Lambda Annotations. Zum Vergleich: Das Verzeichnis `PamApi` enthält äquivalente Dateien, die mit dem regulären Lambda-Programmiermodell geschrieben wurden.

Abhängigkeitsinjektion mit dem Lambda Annotations-Framework

Sie können auch das Lambda Annotations-Framework verwenden, um Ihren Lambda-Funktionen eine Dependency Injection hinzuzufügen, indem Sie die Syntax verwenden, mit der Sie vertraut sind. Wenn Sie einer `Startup.cs` Datei ein `[LambdaStartup]` Attribut hinzufügen, generiert das Lambda Annotations Framework den erforderlichen Code zur Kompilierzeit.

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Ihre Lambda-Funktion kann Dienste entweder durch Konstruktorinjektion oder durch Injektion in einzelne Methoden unter Verwendung des `[FromServices]`-Attributs injizieren.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
```

```
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
        return await this._repo.GetById(id);
    }
}
```

Bewährte Codemethoden für Lambda-Ereignisfunktionen in C#

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die Best Practices für Codierung anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Mit diesem Ansatz können Sie eine Funktion erstellen, die besser anhand von Einheiten getestet werden kann.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda -Ausführungsumgebung enthält eine Reihe von Bibliotheken. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, aktualisiert Lambda diese Bibliotheken regelmäßig. Diese Updates können zu subtilen Änderungen am Verhalten Ihrer Lambda-Funktion führen. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch wird die Zeit reduziert, die benötigt wird, bis Ihr Bereitstellungspaket vor dem Aufruf heruntergeladen und entpackt wird. Vermeiden Sie es bei in .NET verfassten Funktionen, die gesamte AWS SDK-Bibliothek als Teil Ihres Bereitstellungspakets hochzuladen. Verlassen Sie sich stattdessen selektiv auf die Module, die Komponenten des benötigten SDK aufnehmen (z. B. DynamoDB, Amazon S3 SDK-Module und Lambda-Kernbibliotheken).

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

C#-Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven erstellen und bereitstellen

Ein .NET-Bereitstellungspaket (ZIP-Dateiarchiv) enthält die kompilierten Assembly Ihrer Funktion sowie aller ihrer Assembly-Abhängigkeiten. Das Paket enthält auch eine *proj*`.deps.json`-Datei. Dies signalisiert der .NET-Laufzeit alle Abhängigkeiten Ihrer Funktion sowie eine *proj*`.runtimeconfig.json`-Datei, die für die Konfiguration der Laufzeit verwendet wird.

Um einzelne Lambda-Ereignisfunktionen bereitzustellen, können Sie `Amazon.Lambda.Tools` die.NET Lambda Global CLI verwenden. Mit dem `dotnet lambda deploy-function` Befehl wird automatisch ein ZIP-Bereitstellungspaket erstellt und auf Lambda bereitgestellt. Wir empfehlen jedoch, Frameworks wie das AWS Serverless Application Model (AWS SAM) oder das AWS Cloud Development Kit (AWS CDK) zu verwenden, um Ihre .NET-Anwendungen bereitzustellen. AWS

Serverlose Anwendungen bestehen in der Regel aus einer Kombination von Lambda-Funktionen und anderen verwalteten Funktionen, die AWS-Services zusammenarbeiten, um eine bestimmte Geschäftsaufgabe auszuführen. AWS SAM und AWS CDK vereinfachen Sie die Erstellung und Bereitstellung von Lambda-Funktionen mit anderen AWS-Services in großem Maßstab. Die AWS SAM Vorlagenspezifikation bietet eine einfache und übersichtliche Syntax zur Beschreibung von Lambda-Funktionen, APIs, Berechtigungen, Konfigurationen und anderen AWS Ressourcen, aus denen Ihre serverlose Anwendung besteht. Über das AWS CDK definieren Sie die Cloud-Infrastruktur als Code, mit dem Sie zuverlässige, skalierbare und kostengünstige Anwendungen in der Cloud mit modernen Programmiersprachen und Frameworks wie .NET erstellen können. AWS CDK Sowohl die als auch die AWS SAM verwenden das.NET Lambda Global CLI, um Ihre Funktionen zu verpacken.

Obwohl es möglich ist, Lambda-Ebenen mit Funktionen in C# mithilfe der .NET-Core-CLI zu verwenden, raten wir davon ab. Funktionen in C#, die Ebenen verwenden, laden die freigegebenen Baugruppen während des Init-Phase manuell in den Speicher, was die Kaltstartzeiten verlängern kann. Fügen Sie stattdessen den gesamten gemeinsam genutzten Code zur Kompilierungszeit ein, um die Auswirkungen auf die Leistung durch das Laden von Assemblys zur Laufzeit zu vermeiden.

In den folgenden Abschnitten finden Sie Anweisungen zum Erstellen und Bereitstellen von.NET-Lambda-Funktionen mit der AWS SAM AWS CDK, der und der.NET Lambda Global-CLI.

Themen

- Verwenden der.NET Lambda Global CLI
- Stellen Sie C#-Lambda-Ereignisfunktionen bereit mit AWS SAM
- Stellen Sie C#-Lambda-Ereignisfunktionen bereit mit AWS CDK
- Stellen Sie Anwendungen bereit ASP.NET

Verwenden der.NET Lambda Global CLI

Die.NET-CLI und die Erweiterung .NET Lambda Global Tools (`Amazon.Lambda.Tools`) bieten eine plattformübergreifende Möglichkeit zum Erstellen. NET-based Lambda-Anwendungen, packen Sie sie und stellen Sie sie auf Lambda bereit. In diesem Abschnitt erfahren Sie, wie Sie neue Lambda .NET-Projekte mit den .NET CLI- und Amazon Lambda-Vorlagen erstellen und diese mithilfe von `Amazon.Lambda.Tools` verpacken und bereitstellen

Themen

- Voraussetzungen
- .NET-Projekte mit der .NET-CLI erstellen
- .NET-Projekte über die .NET-CLI bereitstellen
- Verwendung von Lambda-Ebenen mit der.NET-CLI

Voraussetzungen

.NET 8-SDK

Falls Sie das noch nicht getan haben, installieren Sie das .NET 8-SDK und Laufzeit.

AWS Amazon.Lambda.Templates .NET-Projektvorlagen

Verwenden Sie das Amazon.Lambda.Templates NuGet Paket, um Ihren Lambda-Funktionscode zu generieren. Zur Installation dieses Vorlagenpakets führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET-Tools für globale CLI

Verwenden Sie zum Erstellen Ihrer Lambda-Funktionen die Amazon.Lambda.Tools-.NET Global Tools-Erweiterung. Um Amazon.Lambda.Tools zu installieren, führen Sie den folgenden Befehl aus:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Weitere Informationen zur Amazon.Lambda.Tools .NET-CLI-Erweiterung finden Sie im AWS Extensions for .NET CLI-Repository unter GitHub.

.NET-Projekte mit der .NET-CLI erstellen

Verwenden Sie in der .NET CLI den `dotnet new`-Befehl zum Erstellen von .NET-Projekten in der Befehlszeile. Lambda bietet mit dem Amazon.Lambda.Templates NuGet Paket zusätzliche Vorlagen an.

Nach der Installation des Pakets führen Sie den folgenden Befehl aus, um eine Liste des verfügbaren Vorlagen zu sehen.

```
dotnet new list
```

Wenn Sie Details zu einer Vorlage einsehen möchten, verwenden Sie die `help`-Option. Um beispielsweise Details über die `lambda.EmptyFunction`-Vorlage zu sehen, führen Sie den folgenden Befehl aus.

```
dotnet new lambda.EmptyFunction --help
```

Verwenden Sie die `lambda.EmptyFunction`-Vorlage, um eine Basisvorlage für eine .NET-Lambda-Funktion zu erstellen. Dadurch wird eine einfache Funktion erstellt, die eine Zeichenfolge als Eingabe verwendet und diese mithilfe der `ToUpper`-Methode in Großbuchstaben umwandelt. Diese Vorlage unterstützt die folgenden Optionen.

- `--name` – Der Name der Funktion.
- `--region`— Die AWS Region, in der die Funktion erstellt werden soll.
- `--profile`— Der Name eines Profils in Ihrer AWS SDK für .NET Anmeldeinformationsdatei. Weitere Informationen zu Anmeldeinformationsprofilen in.NET finden Sie unter Configure AWS Credentials im AWS SDK for .NET Developer Guide.

In diesem Beispiel erstellen wir eine neue leere Funktion `myDotnetFunction` mit dem Namen unter Verwendung des Standardprofils und der AWS-Region Standardeinstellungen:

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

Mit diesem Befehl werden die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis erstellt.

```
### myDotnetFunction
```

```
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

Zeigen Sie im Verzeichnis `src/myDotnetFunction` die folgenden Dateien an:

- aws-lambda-tools-defaults.json: Hier geben Sie die Befehlszeilenoptionen an, wenn Sie Ihre Lambda-Funktion bereitstellen. Beispiel:

```
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
  "function-architecture": "x86_64",
  "function-runtime":"dotnet8",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
```

- Function.cs: Ihr Lambda-Handler-Funktionscode. Dies ist eine C #-Vorlage mit der `Amazon.Lambda.Core`-Standardbibliothek und einem `LambdaSerializer`-Standardattribut. Weitere Informationen zu den Serialisierungsanforderungen und -optionen finden Sie unter Serialisierung in C#-Lambda-Ereignisfunktionen. Es ist auch eine Beispielfunktion enthalten, die Sie bearbeiten können, um Lambda-Funktionscode anzuwenden.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
 a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

namespace myDotnetFunction;

public class Function
{
```

```
    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- my DotnetFunction.csproj: Eine MSBuild-Datei, die die Dateien und Assemblys auflistet, aus denen Ihre Anwendung besteht.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- Readme: Verwenden Sie diese Datei, um Ihre Lambda-Funktion zu dokumentieren.

Zeigen Sie im Verzeichnis `myfunction/test` die folgenden Dateien an:

- my DotnetFunction.Tests.csproj: Wie bereits erwähnt, ist dies eine MSBuild-Datei, die die Dateien und Assemblys auflistet, aus denen Ihr Testprojekt besteht. Beachten Sie auch, dass die `Amazon.Lambda.Core`-Bibliothek enthalten ist. Damit können Sie nahtlos beliebige Lambda-Vorlagen integrieren, die zum Testen Ihrer Funktion erforderlich sind.

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs: Dieselbe C#-Code-Vorlagendatei, in der sie im `src` Verzeichnis enthalten ist. Bearbeiten Sie diese Datei, damit sie dem Produktionscode Ihrer Funktion gleicht, und testen Sie sie vor dem Hochladen Ihrer Lambda-Funktion in eine Produktionsumgebung.

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

.NET-Projekte über die .NET-CLI bereitstellen

Um Ihr Bereitstellungspaket zu erstellen und es auf Lambda bereitzustellen, verwenden Sie die `Amazon.Lambda.Tools`-CLI-Tools. Um Ihre Funktion anhand der Dateien bereitzustellen, die Sie in den vorherigen Schritten erstellt haben, navigieren Sie zunächst zu dem Ordner, der die `.csproj`-Datei Ihrer Funktion enthält.

```
cd myDotnetFunction/src/myDotnetFunction
```

Führen Sie den folgenden Befehl aus, um Ihren Code als ZIP-Bereitstellungspaket für Lambda bereitzustellen. Wählen Sie Ihren eigenen Funktionsnamen.

```
dotnet lambda deploy-function myDotnetFunction
```

Während der Bereitstellung fordert Sie der Assistent auf, einen the section called “Ausführungsrolle (Berechtigungen für Funktionen zum Zugriff auf andere Ressourcen)” auszuwählen. Wählen Sie für dieses Beispiel die `lambda_basic_role`.

Nachdem Sie Ihre Funktion bereitgestellt haben, können Sie sie mit dem `dotnet lambda invoke-function`-Befehl in der Cloud testen. Für den Beispielcode in der `lambda.EmptyFunction`-Vorlage können Sie Ihre Funktion testen, indem Sie mit der `--payload`-Option eine Zeichenfolge übergeben.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

Wenn Ihre Funktion erfolgreich bereitgestellt wurde, sollten Sie eine Ausgabe ähnlich der folgenden sehen.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
```

```
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

Verwendung von Lambda-Ebenen mit der.NET-CLI

> **Note**
>
> Es ist zwar möglich, Ebenen mit Funktionen in .NET zu verwenden, wir raten jedoch davon ab. Funktionen in .NET, die Ebenen verwenden, laden die gemeinsam genutzten Assemblys während der `Init`-Phase manuell in den Speicher, was die Kaltstartzeiten verlängern kann. Schließen Sie stattdessen den gesamten freigegebenen Code zur Kompilierzeit ein, um die integrierten Optimierungen des .NET-Compilers zu nutzen.

Die .NET-CLI unterstützt Befehle, mit denen Sie Ebenen veröffentlichen und C#-Funktionen bereitstellen können, die Ebenen nutzen. Um eine Ebene in einem angegebenen Amazon-S3-Bucket zu veröffentlichen, führen Sie den folgenden Befehl im selben Verzeichnis wie Ihre `.csproj`-Datei aus:

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

Wenn Sie dann Ihre Funktion mithilfe der .NET-CLI bereitstellen, geben Sie im folgenden Befehl den Ebenen-ARN an, den Sie verwenden wollen:

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

Ein vollständiges Beispiel einer Hello-World-Funktion finden Sie im Beispiel blank-csharp-with-layer.

Stellen Sie C#-Lambda-Ereignisfunktionen bereit mit AWS SAM

The AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode

und allen von Ihnen angegebenen Abhängigkeiten erstellt. AWS SAM stellt dann Ihre Funktion mithilfe eines Stacks bereit.CloudFormation Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Die folgenden Schritte zeigen Ihnen, wie Sie eine .NET Hello World-Beispielanwendung mit AWS SAM herunterladen, erstellen und bereitstellen können. Diese Beispielanwendung verwendet eine Lambda-Funktion und einen Amazon API Gateway-Endpunkt, um ein grundlegendes API-Backend zu implementieren. Wenn Sie eine HTTP GET-Anforderung an Ihren API Gateway-Endpunkt senden, ruft API Gateway Ihre Lambda-Funktion auf. Die Funktion gibt eine „Hello World“ -Meldung zusammen mit der IP-Adresse der Lambda-Funktionsinstance zurück, die Ihre Anfrage verarbeitet.

Wenn Sie Ihre Anwendung mithilfe von erstellen und bereitstellen AWS SAM, verwendet die AWS SAM CLI im Hintergrund den `dotnet lambda package` Befehl, um die einzelnen Lambda-Funktionscodepakete zu verpacken.

Voraussetzungen

.NET 8-SDK

Installieren Sie das .NET 8-SDK und Runtime.

AWS SAM CLI Version 1.39 oder höher

Informationen zur Installation der neuesten Version der AWS SAM CLI finden Sie unter Installation der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mit der Hello World-.NET-Vorlage mit dem folgenden Befehl.

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

Mit diesem Befehl werden die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis erstellt.

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
```

```
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
    #       ### Function.cs
    #       ### HelloWorld.csproj
    #       ### aws-lambda-tools-defaults.json
    ### template.yaml
    ### test
        ### HelloWorld.Test
            ### FunctionTest.cs
            ### HelloWorld.Tests.csproj
```

2. Navigieren Sie in das Verzeichnis, das die `template.yaml file` enthält. Diese Datei ist eine Vorlage, die die AWS -Ressourcen für Ihre Anwendung definiert, einschließlich Ihrer Lambda-Funktion und einer API-Gateway-API.

```
cd sam-app
```

3. Um den Quellcode Ihrer Anwendung zu erstellen, führen Sie folgenden Befehl aus.

```
sam build
```

4. Führen Sie den folgenden Befehl aus AWS, um Ihre Anwendung bereitzustellen.

```
sam deploy --guided
```

Dieser Befehl packt Ihre Anwendung und stellt sie mit der folgenden Reihe von Eingabeaufforderungen bereit. Um die Standardoptionen zu übernehmen, drücken Sie die Eingabetaste.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert, ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

- Stack-Name: Der Name des Stacks, der in CloudFormation bereitgestellt werden soll. Dieser Name muss für Ihr AWS-Konto Land eindeutig sein AWS-Region.
- AWS-Region: Die AWS-Region , auf der Sie Ihre App bereitstellen möchten.

- Änderungen vor der Bereitstellung bestätigen: Wählen Sie Ja, um alle Änderungssätze manuell zu überprüfen, bevor AWS SAM die Anwendungsänderungen bereitstellt. Wenn Sie Nein wählen, stellt die AWS SAM CLI automatisch Anwendungsänderungen bereit.
- Erlauben Sie die Erstellung von SAM-CLI-IAM-Rollen: Viele AWS SAM Vorlagen, in diesem Beispiel die von Hello World, erstellen AWS Identity and Access Management (IAM-) Rollen, um Ihren Lambda-Funktionen Zugriff auf andere zu gewähren. AWS-Services Wählen Sie Ja aus, um die Erlaubnis zur Bereitstellung eines CloudFormation Stacks zu erteilen, der IAM-Rollen erstellt oder ändert.
- Rollback deaktivieren: Wenn bei der Erstellung oder Bereitstellung Ihres Stacks ein Fehler auftritt, wird der Stack standardmäßig AWS SAM auf die vorherige Version zurückgesetzt. Wählen Sie Nein, um diese Standardeinstellung zu akzeptieren.
- HelloWorldFunction Möglicherweise ist keine Autorisierung definiert, ist das in Ordnung: Geben Sie einy.
- Argumente in samconfig.toml speichern: Wählen Sie Ja, um Ihre Konfigurationsauswahl zu speichern. In Zukunft können Sie `sam deploy` ohne Parameter erneut ausführen, um Änderungen an Ihrer Anwendung vorzunehmen.

5. Wenn die Bereitstellung Ihrer Anwendung abgeschlossen ist, gibt die CLI den Amazon Resource Name (ARN) der Hello World Lambda-Funktion und die für sie erstellte IAM-Rolle zurück. Sie zeigt auch den Endpunkt Ihrer API-Gateway-API an. Um Ihre Anwendung zu testen, öffnen Sie den Endpunkt in einem Browser. Es wird eine Antwort ähnlich der folgenden angezeigt.

```
{"message":"hello world","location":"34.244.135.203"}
```

6. Um Ihre Ressourcen zu löschen, führen Sie den folgenden Befehl aus. Beachten Sie, dass der von Ihnen erstellte API-Endpunkt ein öffentlicher Endpunkt ist, auf den über das Internet zugegriffen werden kann. Es wird empfohlen, dass Sie diesen Endpunkt nach dem Testen löschen.

```
sam delete
```

Nächste Schritte

Weitere Informationen AWS SAM zur Erstellung und Bereitstellung von Lambda-Funktionen mit.NET finden Sie in den folgenden Ressourcen:

- Das AWS Serverless Application Model (AWS SAM) Entwicklerhandbuch

- Serverlose .NET-Anwendungen mit AWS Lambda und der SAM-CLI CLI

Stellen Sie C#-Lambda-Ereignisfunktionen bereit mit AWS CDK

Das AWS Cloud Development Kit (AWS CDK) ist ein Open-Source-Framework für die Softwareentwicklung zur Definition von Cloud-Infrastruktur als Code mit modernen Programmiersprachen und Frameworks wie.NET. AWS CDK Projekte werden ausgeführt, um CloudFormation Vorlagen zu generieren, die dann zur Bereitstellung Ihres Codes verwendet werden.

Folgen Sie den Anweisungen in den folgenden Abschnitten AWS CDK, um eine Hello World-.NET-Beispielanwendung mit dem zu erstellen und bereitzustellen. Die Beispielanwendung implementiert ein grundlegendes API-Backend, das aus einem API-Gateway-Endpunkt und einer Lambda-Funktion besteht. API Gateway ruft die Lambda-Funktion auf, wenn Sie eine HTTP-GET-Anforderung an den Endpunkt senden. Die Funktion gibt eine Hello-World-Nachricht zusammen mit der IP-Adresse der Lambda-Instance zurück, die Ihre Anfrage verarbeitet.

Voraussetzungen

.NET 8-SDK

Installieren Sie das .NET 8-SDK und Runtime.

AWS CDK Version 2

Informationen zur Installation der neuesten Version von finden Sie AWS CDK unter Erste Schritte mit dem AWS CDK im AWS Cloud Development Kit (AWS CDK) v2-Entwicklerhandbuch.

Stellen Sie ein Beispiel bereit AWS CDK Anwendung

1. Erstellen Sie ein Projektverzeichnis für die Beispielanwendung und navigieren Sie dorthin.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie eine neue AWS CDK Anwendung, indem Sie den folgenden Befehl ausführen.

```
cdk init app --language csharp
```

Der Befehl erstellt die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. Öffnen Sie das `src`-Verzeichnis und erstellen Sie eine neue Lambda-Funktion mit der.NET-CLI. Dies ist die Funktion, die Sie mit dem AWS CDK bereitstellen. In diesem Beispiel erstellen Sie eine Hello world-Funktion mit dem Namen `HelloWorldLambda` unter Verwendung der `lambda.EmptyFunction`-Vorlage.

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

Nach diesem Schritt sollte Ihre Verzeichnisstruktur innerhalb Ihres Projektverzeichnisses wie folgt aussehen.

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
```

```
                ### HelloWorldLambda.Tests.csproj
```

4. Öffnen Sie die `HelloWorldStack.cs`-Datei aus dem Verzeichnis `src/HelloWorld`. Ersetzen Sie den Inhalt der Datei durch den folgenden Code.

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
                " && dotnet build"+
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };

             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
```

```
                    Bundling = buildOption
                }),
            });
        }
    }
}
```

Dies ist der Code zum Kompilieren und Bündeln des Anwendungscodes sowie die Definition der Lambda-Funktion selbst. Mit dem `BundlingOptions`-Objekt kann eine Zip-Datei erstellt werden, zusammen mit einer Reihe von Befehlen, die zur Erzeugung des Inhalts der Zip-Datei verwendet werden. In diesem Fall wird der Befehl `dotnet lambda package` zum Kompilieren und Erzeugen der Zip-Datei verwendet.

5. Um Ihre Anwendung bereitzustellen, führen Sie den folgenden Befehl aus.

```
cdk deploy
```

6. Rufen Sie Ihre bereitgestellte Lambda-Funktion mit der .NET Lambda CLI auf.

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. Nach Abschluss der Tests können Sie die erstellten Ressourcen löschen, es sei denn, Sie möchten sie beibehalten. Führen Sie den folgenden Befehl aus, um Ihre Ressourcen zu löschen.

```
cdk destroy
```

Nächste Schritte

Weitere Informationen AWS CDK zur Erstellung und Bereitstellung von Lambda-Funktionen mit.NET finden Sie in den folgenden Ressourcen:

- Arbeiten mit dem AWS CDK in C#
- Erstellen, Verpacken und Veröffentlichen von.NET C#-Lambda-Funktionen mit dem CDK AWS

Stellen Sie Anwendungen bereit ASP.NET

Sie können nicht nur ereignisgesteuerte Funktionen hosten, sondern auch .NET mit Lambda Event Functions verwenden, um einfache Anwendungen zu hosten. ASP.NET Mit dem Paket können Sie ASP.NET Anwendungen erstellen und bereitstellen. `Amazon.Lambda.AspNetCoreServer` NuGet

In diesem Abschnitt erfahren Sie, wie Sie mithilfe des.NET Lambda-CLI-Tools eine ASP.NET Web-API für Lambda bereitstellen.

Themen

- Voraussetzungen
- Bereitstellen einer ASP.NET Web-API für Lambda
- Bereitstellung ASP.NET minimaler APIs für Lambda

Voraussetzungen

.NET 8-SDK

Installieren Sie das.NET 8 SDK und ASP.NET Core Runtime.

Amazon.Lambda.Tools

Verwenden Sie zum Erstellen Ihrer Lambda-Funktionen die Amazon.Lambda.Tools-.NET Global Tools-Erweiterung. Um Amazon.Lambda.Tools zu installieren, führen Sie den folgenden Befehl aus:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Weitere Informationen zur Amazon.Lambda.Tools .NET-CLI-Erweiterung finden Sie im AWS Extensions for .NET CLI-Repository unter GitHub.

Amazon.Lambda.Templates

Verwenden Sie das Amazon.Lambda.Templates NuGet Paket, um Ihren Lambda-Funktionscode zu generieren. Zur Installation dieses Vorlagenpakets führen Sie den folgenden Befehl aus:

```
dotnet new --install Amazon.Lambda.Templates
```

Bereitstellen einer ASP.NET Web-API für Lambda

Um eine Web-API bereitzustellen ASP.NET, können Sie die .NET Lambda-Vorlagen verwenden, um ein neues Web-API-Projekt zu erstellen. Verwenden Sie den folgenden Befehl, um ein neues ASP.NET Web-API-Projekt zu initialisieren. Im Beispielbefehl nennen wir das Projekt `AspNetOnLambda`.

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

Mit diesem Befehl werden die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis erstellt.

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

Wenn Lambda Ihre Funktion aufruft, wird als Einstiegspunkt die Datei `LambdaEntryPoint.cs` verwendet. Die von der .NET Lambda-Vorlage erstellte Datei enthält den folgenden Code.

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
```

```
    {
    }
}
```

Der von Lambda verwendete Einstiegspunkt muss von einer der drei Basisklassen im Paket `Amazon.Lambda.AspNetCoreServer` erben. Diese drei Basisklassen sind:

- `APIGatewayProxyFunction`
- `APIGatewayHttpApiV2ProxyFunction`
- `ApplicationLoadBalancerFunction`

Die Standardklasse, die verwendet wird, wenn Sie Ihre `LambdaEntryPoint.cs`-Datei mit der bereitgestellten .NET Lambda-Vorlage erstellen, ist `APIGatewayProxyFunction`. Die Basisklasse, die Sie in Ihrer Funktion verwenden, hängt davon ab, welche API-Schicht vor Ihrer Lambda-Funktion steht.

Jede der drei Basisklassen enthält eine öffentliche Methode namens `FunctionHandlerAsync`. Der Name dieser Methode ist Teil der Handlerzeichenfolge, die Lambda verwendet, um Ihre Funktion aufzurufen. Die `FunctionHandlerAsync` Methode wandelt die Nutzlast für eingehende Ereignisse in das richtige ASP.NET Format und die ASP.NET Antwort zurück in eine Lambda-Antwortnutzlast. Für das gezeigte Beispielprojekt `AspNetOnLambda` würde der Handler-String wie folgt lauten.

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

Um die API in Lambda bereitzustellen, führen Sie die folgenden Befehle aus, um in das Verzeichnis mit Ihrer Quellcodedatei zu navigieren und Ihre Funktion mit CloudFormation bereitzustellen.

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> Wenn Sie eine API mithilfe des **`dotnet lambda deploy-serverless`** Befehls bereitstellen, CloudFormation gibt es Ihrer Lambda-Funktion einen Namen, der auf dem Stacknamen basiert, den Sie während der Bereitstellung angeben. Um Ihrer Lambda-Funktion einen benutzerdefinierten Namen zu geben, bearbeiten Sie die `serverless.template`-Datei, um der `AWS::Serverless::Function`-Ressource

eine `FunctionName`-Eigenschaft hinzuzufügen. Weitere Informationen finden Sie unter Namenstyp im CloudFormation -Benutzerhandbuch.

Bereitstellung ASP.NET minimaler APIs für Lambda

Um eine ASP.NET Minimal-API für Lambda bereitzustellen, können Sie die.NET-Lambda-Vorlagen verwenden, um ein neues Minimal-API-Projekt zu erstellen. Verwenden Sie den folgenden Befehl, um ein neues minimales API-Projekt zu initialisieren. In diesem Beispiel nennen wir das Projekt `MinimalApiOnLambda`.

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

Der Befehl erstellt die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis.

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

Die Datei `Program.cs` enthält den folgenden Code.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);
```

```
var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

Um Ihre Minimal-API für die Ausführung auf Lambda zu konfigurieren, müssen Sie diesen Code möglicherweise bearbeiten, damit Anfragen und Antworten zwischen Lambda und ASP.NET Core korrekt übersetzt werden. Standardmäßig ist die Funktion für eine REST-API-Ereignisquelle konfiguriert. Für eine HTTP-API oder einen Application Load Balancer ersetzen Sie `(LambdaEventSource.RestApi)` durch eine der folgenden Optionen:

- `(LambdaEventSource.HttpAPi)`
- `(LambdaEventSource.ApplicationLoadBalancer)`

Um Ihre Minimal-API in Lambda bereitzustellen, führen Sie die folgenden Befehle aus, um in das Verzeichnis mit Ihrer Quellcodedatei zu navigieren und Ihre Funktion mit CloudFormation bereitzustellen.

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

Arbeiten mit Layern für.NET Lambda Event Functions

Es wird nicht empfohlen, Ebenen zur Verwaltung von Abhängigkeiten für in .NET geschriebene Lambda-Ereignisfunktionen zu verwenden. Der Grund dafür ist, dass .NET eine kompilierte Sprache ist und Ihre Funktionen während der Init-Phase trotzdem alle gemeinsam genutzten Assemblys manuell in den Speicher laden müssen, was die Kaltstartzeiten verlängern kann. Die Verwendung von Ebenen verkompliziert nicht nur den Bereitstellungsprozess, sondern verhindert auch, dass Sie die Vorteile der integrierten Compiler-Optimierungen nutzen können.

Um externe Abhängigkeiten mit Ihren .NET-Handlern zu verwenden, fügen Sie diese bei der Kompilierung direkt in Ihr Bereitstellungspaket ein. So vereinfachen Sie den Bereitstellungsprozess und können außerdem die integrierten .NET-Compiler-Optimierungen nutzen. Ein Beispiel dafür, wie Sie Abhängigkeiten wie NuGet Pakete in Ihrer Funktion importieren und verwenden, finden Sie unterthe section called “Handler”.

Bereitstellen von.NET-Lambda-Ereignisfunktionen mit Container-Images

Es gibt drei Möglichkeiten, ein Container-Image für eine.NET-Lambda-Ereignisfunktion zu erstellen:

- Mit einem AWS Basisimage für.NET

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Mit einem AWS OS-only Basisbild

  AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für .NET in das Image aufnehmen.

- Verwenden eines AWS Nicht-Basis-Images

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für .NET in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basisimages für.NET
- Mit einem AWS Basisimage für.NET

- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basisimages für.NET

AWS stellt die folgenden Basisimages für.NET bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
10	.NET 10	Amazon Linux 2023	Dockerfile für .NET 10 auf GitHub	14. November 2028
9	.NET 9	Amazon Linux 2023	Dockerfile für .NET 9 auf GitHub	10. November 2026
8	.NET 8	Amazon Linux 2023	Dockerfile für .NET 8 auf GitHub	10. November 2026

Amazon ECR-Repository: gallery.ecr. aws/lambda/dotnet

Mit einem AWS Basisimage für.NET

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET SDK – Die folgenden Schritte verwenden das .NET 8-Basis-Image. Stellen Sie sicher, dass Ihre .NET-Version mit der Version des Basis-Images übereinstimmt, die Sie in Ihrer Docker-Datei angeben.
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen und Bereitstellen eines Images mithilfe eines Basis-Images

In den folgenden Schritten verwenden Sie Amazon.Lambda.Templatesund, Amazon.Lambda.Toolsum ein .NET-Projekt zu erstellen. Anschließend erstellen Sie ein Docker-Image, laden das Image auf Amazon ECR hoch und stellen es auf einer Lambda-Funktion bereit.

1. Installieren Sie das Paket Amazon.Lambda.Templates NuGet .

```
dotnet new install Amazon.Lambda.Templates
```

2. Erstellen Sie ein .NET-Projekt mithilfe der `lambda.image.EmptyFunction`-Vorlage.

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

Die Projektdateien werden im *`MyFunction`*`/src/`*`MyFunction`*-Verzeichnis gespeichert:

- aws-lambda-tools-defaults.json: Gibt die Befehlszeilenoptionen zum Bereitstellen Ihrer Lambda-Funktion an.
- Function.cs: Ihr Lambda-Handler-Funktionscode. Dies ist eine C#-Vorlage mit der `Amazon.Lambda.Core`-Standardbibliothek und einem `LambdaSerializer`-Standardattribut. Weitere Informationen über Serialisierungsanforderungen und -optionen finden Sie unter Serialisierung in C#-Lambda-Ereignisfunktionen. Sie können den bereitgestellten Code zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.
- MyFunction.csproj: Eine .NET-Projektdatei, die die Dateien und Assemblys auflistet, aus denen Ihre Anwendung besteht.
- Dockerfile: Sie können die bereitgestellte Docker-Datei zum Testen verwenden oder sie durch Ihre eigene ersetzen. Wenn Sie Ihre eigenen verwenden, stellen Sie sicher, dass Sie:
  - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images. Das Basis-Image und das `TargetFramework` in der Datei `MyFunction.csproj` müssen beide dieselbe .NET-Version verwenden. Um beispielsweise .NET 9 zu verwenden:
    - Docker-Datei: `FROM` *`public.ecr.aws/lambda/dotnet:9`*
    - MyFunction.csproj: `<TargetFramework>`*`net9.0`*`</TargetFramework>`
  - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest. Das sollte mit dem `image-command` in `aws-lambda-tools-defaults.json` übereinstimmen.
3. Installieren Sie Amazon.Lambda.Tools das.NET Global Tool.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Wenn Amazon.Lambda.Tools es bereits installiert ist, stellen Sie sicher, dass Sie die neueste Version haben.

```
dotnet tool update -g Amazon.Lambda.Tools
```

4. Ändern Sie das Verzeichnis zu *MyFunction*/src/*MyFunction*, sofern Sie dies noch nicht getan haben.

```
cd src/MyFunction
```

5. Wird verwendet Amazon.Lambda.Tools , um das Docker-Image zu erstellen, es in ein neues Amazon ECR-Repository zu übertragen und die Lambda-Funktion bereitzustellen.

   Für `--function-role` geben Sie den Rollennamen – nicht den Amazon-Ressourcenname (ARN) – der Ausführungsrolle für die Funktion an. Beispiel, `lambda-role`.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

   Weitere Informationen zum Amazon.Lambda.Tools .NET Global Tool finden Sie im AWS Extensions for .NET CLI Repository unter GitHub.

6. Die Funktion aufrufen.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

   Wenn alles erfolgreich ist, sehen Sie eine Antwort ähnlich der folgenden:

```
Payload:
{"Lower":"testing the function","Upper":"TESTING THE FUNCTION"}

Log Tail:
INIT_REPORT Init Duration: 9999.81 ms   Phase: init     Status: timeout
START RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed Version: $LATEST
END RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed
REPORT RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed  Duration: 3173.06 ms
 Billed Duration: 3174 ms        Memory Size: 512 MB     Max Memory Used: 24 MB
```

7. Löschen Sie die Lambda-Funktion.

```
dotnet lambda delete-function MyFunction
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-only Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Runtime-Interface-Client in Ihr Image aufnehmen. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Das folgende Beispiel zeigt, wie ein Container-Image für .NET mithilfe eines AWS Nicht-Basis-Images erstellt wird und wie das Amazon.Lambda.RuntimeSupport Paket hinzugefügt wird, das der Lambda-Runtime-Interface-Client für .NET ist. Das Beispiel-Dockerfile verwendet das Basis-Image „Microsoft .NET 8“.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET SDK — In den folgenden Schritten wird ein .NET 9-Basisimage verwendet. Stellen Sie sicher, dass Ihre .NET-Version mit der Version des Basis-Images übereinstimmt, die Sie in Ihrer Docker-Datei angeben.
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen und Bereitstellen eines Images mithilfe eines alternativen Basis-Images

1. Installieren Sie das Paket Amazon.Lambda.Templates NuGet .

   ```
   dotnet new install Amazon.Lambda.Templates
   ```

2. Erstellen Sie ein .NET-Projekt mithilfe der `lambda.CustomRuntimeFunction`-Vorlage. Diese Vorlage enthält das Amazon.Lambda.RuntimeSupportPaket.

   ```
   dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
   ```

3. Navigieren Sie zum *`MyFunction`*`/src/`*`MyFunction`* Verzeichnis . Hier werden die Projektdateien gespeichert. Prüfen Sie die folgenden Dateien:

   - aws-lambda-tools-defaults.json – In dieser Datei geben Sie die Befehlszeilenoptionen an, wenn Sie Ihre Lambda-Funktion bereitstellen.
   - Function.cs— Der Code enthält eine Klasse mit einer `Main` Methode, die die `Amazon.Lambda.RuntimeSupport` Bibliothek als Bootstrap initialisiert. Die `Main`-

Methode ist der Einstiegspunkt für den Prozess der Funktion. Die `Main`-Methode verpackt den Funktionshandler in einen Wrapper, mit dem der Bootstrap arbeiten kann. Weitere Informationen finden Sie unter Amazon.Lambda.RuntimeSupport Als Klassenbibliothek im Repository verwenden. GitHub

- MyFunction.csproj— Eine .NET-Projektdatei, die die Dateien und Assemblys auflistet, aus denen Ihre Anwendung besteht.
- Readme.md— Diese Datei enthält weitere Informationen zur Lambda-Beispielfunktion.

4. Öffnen Sie die Datei `aws-lambda-tools-defaults.json` und fügen Sie die folgenden Zeilen hinzu:

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

- package-type: Definiert das Bereitstellungspaket als Container-Image.
- docker-host-build-output-dir: Legt das Ausgabeverzeichnis für den Build-Prozess fest.

Example aws-lambda-tools-defaults.json

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. Erstellen Sie ein Dockerfile in Verzeichnis *MyFunction*/src/*MyFunction*. Das folgende Beispiel-Dockerfile verwendet ein Microsoft .NET-Basisimage anstelle eines AWS -Basis-Images.

   - Legen Sie FROM-Eigenschaft auf die Kennung des Basis-Images fest. Das Basis-Image und das `TargetFramework` in der Datei `MyFunction.csproj` müssen beide dieselbe .NET-Version verwenden.
   - Verwenden Sie den COPY-Befehl, um die Funktion in das Verzeichnis `/var/task` zu kopieren.
   - Legen Sie ENTRYPOINT auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Bootstrap, der die Bibliothek `Amazon.Lambda.RuntimeSupport` initialisiert.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

   Example Dockerfile

   ```
   # You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
   FROM mcr.microsoft.com/dotnet/runtime:9.0

   # Set the image's internal work directory
   WORKDIR /var/task

   # Copy function code to Lambda-defined environment variable
   COPY "bin/Release/net9.0/linux-x64"  .

   # Set the entrypoint to the bootstrap
   ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
   ```

6. Installieren Sie die Erweiterung Amazon.Lambda.Tools .NET Global Tools.

   ```
   dotnet tool install -g Amazon.Lambda.Tools
   ```

   Wenn sie bereits installiert Amazon.Lambda.Tools ist, stellen Sie sicher, dass Sie die neueste Version haben.

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Wird verwendet Amazon.Lambda.Tools , um das Docker-Image zu erstellen, es in ein neues Amazon ECR-Repository zu übertragen und die Lambda-Funktion bereitzustellen.

   Für `--function-role` geben Sie den Rollennamen – nicht den Amazon-Ressourcenname (ARN) – der Ausführungsrolle für die Funktion an. Beispiel, `lambda-role`.

   ```
   dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   Weitere Informationen zur Amazon.Lambda.Tools .NET-CLI-Erweiterung finden Sie im AWS Extensions for .NET CLI-Repository unter GitHub.

8. Die Funktion aufrufen.

   ```
   dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   Wenn alles erfolgreich ist, sehen Sie Folgendes:

   ```
   Payload:
   "TESTING THE FUNCTION"

   Log Tail:
   START RequestId: id Version: $LATEST
   END RequestId: id
   REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
    Size: 256 MB     Max Memory Used: 12 MB
   ```

9. Löschen Sie die Lambda-Funktion.

   ```
   dotnet lambda delete-function MyFunction
   ```

Kompilieren Sie.NET-Lambda-Ereignisfunktionscode in ein natives Laufzeitformat

.NET 8 unterstützt die native Ahead-of-Time-Kompilierung (AOT). Mit nativem AOT können Sie Ihren Lambda-Funktionscode in ein natives Laufzeitformat kompilieren, wodurch die Notwendigkeit entfällt, .NET-Code zur Laufzeit zu kompilieren. Die native AOT-Kompilierung kann die Kaltstartzeit für Lambda-Funktionen reduzieren, die Sie in .NET schreiben. Weitere Informationen finden Sie im AWS Compute-Blog unter Einführung in die .NET 8-Laufzeit für AWS Lambda.

Sections

- Lambda-Laufzeit
- Voraussetzungen
- Erste Schritte
- Serialisierung
- Trimmen
- Fehlerbehebung

Lambda-Laufzeit

Um eine Lambda-Funktion mit nativer AOT-Kompilierung zu erstellen, verwenden Sie die verwaltete .NET 8 Lambda-Laufzeit. Diese Laufzeit unterstützt die Verwendung sowohl der x86_64- als auch arm64-Architektur.

Wenn Sie eine .NET Lambda-Funktion ohne AOT bereitstellen, wird Ihre Anwendung zunächst in IL-Code (Intermediate Language) kompiliert. Zur Laufzeit nimmt der Just-in-Time-Compiler (JIT) in der Lambda-Laufzeitumgebung den IL-Code und kompiliert ihn nach Bedarf in Maschinencode. Mit einer Lambda-Funktion, die im Voraus mit nativem AOT kompiliert wird, kompilieren Sie Ihren Code bei der Bereitstellung Ihrer Funktion in Maschinencode, sodass Sie nicht auf die .NET-Laufzeit oder das SDK in der Lambda-Laufzeit angewiesen sind, um Ihren Code vor der Ausführung zu kompilieren.

Eine Einschränkung von AOT ist, dass Ihr Anwendungscode in einer Umgebung mit demselben Amazon Linux 2023 (AL2023) Betriebssystem kompiliert werden muss, das auch die .NET 8-Laufzeitumgebung verwendet. Die .NET Lambda-CLI bietet Funktionen zum Kompilieren Ihrer Anwendung in einem Docker-Container mithilfe eines AL2023-Images.

Um mögliche Probleme mit der architekturübergreifenden Kompatibilität zu vermeiden, empfehlen wir dringend, dass Sie Ihren Code in einer Umgebung mit derselben Prozessorarchitektur kompilieren, die Sie für Ihre Funktion konfigurieren. Weitere Informationen zu den Einschränkungen

der architekturübergreifenden Kompilierung finden Sie Cross-compilationin der Microsoft.NET-Dokumentation.

Voraussetzungen

Docker

Um natives AOT zu verwenden, muss Ihr Funktionscode in einer Umgebung mit demselben AL2023-Betriebssystem wie die .NET 8-Laufzeit kompiliert werden. Die .NET-CLI-Befehle in den folgenden Abschnitten verwenden Docker, um Lambda-Funktionen in einer AL2023-Umgebung zu entwickeln und zu erstellen.

.NET 8-SDK

Die native AOT-Kompilierung ist ein Feature von .NET 8. Sie müssen das .NET 8-SDK auf Ihrer Build-Machine installieren, nicht nur die Laufzeit.

Amazon.Lambda.Tools

Verwenden Sie zum Erstellen Ihrer Lambda-Funktionen die Amazon.Lambda.Tools-.NET Global Tools-Erweiterung. Um Amazon.Lambda.Tools zu installieren, führen Sie den folgenden Befehl aus:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Weitere Informationen zur Amazon.Lambda.Tools .NET-CLI-Erweiterung finden Sie im AWS Extensions for .NET CLI-Repository unter GitHub.

Amazon.Lambda.Templates

Verwenden Sie das Amazon.Lambda.Templates NuGet Paket, um Ihren Lambda-Funktionscode zu generieren. Zur Installation dieses Vorlagenpakets führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

Erste Schritte

Sowohl das.NET Global CLI als auch AWS Serverless Application Model (AWS SAM) bieten Vorlagen für erste Schritte zum Erstellen von Anwendungen mit nativem AOT. Um Ihre erste native AOT-Lambda-Funktion zu erstellen, führen Sie die Schritte in den folgenden Anweisungen aus.

So initialisieren und implementieren Sie eine native AOT-kompilierte Lambda-Ereignisfunktion

1. Initialisieren Sie ein neues Projekt unter Verwendung der nativen AOT-Vorlage und navigieren Sie dann in das Verzeichnis, das die erstellten `.cs`- und `.csproj`-Dateien enthält. In diesem Beispiel nennen wir unsere Funktion `NativeAotSample`.

   ```
   dotnet new lambda.NativeAOT -n NativeAotSample
   cd ./NativeAotSample/src/NativeAotSample
   ```

   Die von der nativen AOT-Vorlage erstellte `Function.cs`-Datei enthält den folgenden Funktionscode.

   ```
   using Amazon.Lambda.Core;
   using Amazon.Lambda.RuntimeSupport;
   using Amazon.Lambda.Serialization.SystemTextJson;
   using System.Text.Json.Serialization;

   namespace NativeAotSample;

   public class Function
   {
       /// <summary>
       /// The main entry point for the Lambda function. The main function is called
    once during the Lambda init phase. It
       /// initializes the .NET Lambda runtime client passing in the function handler
    to invoke for each Lambda event and
       /// the JSON serializer to use for converting Lambda JSON format to the .NET
    types.
       /// </summary>
       private static async Task Main()
       {
           Func<string, ILambdaContext, string> handler = FunctionHandler;
           await LambdaBootstrapBuilder.Create(handler, new
    SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
               .Build()
               .RunAsync();
       }

       /// <summary>
       /// A simple function that takes a string and does a ToUpper.
       ///
   ```

```
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
```

```
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

Native AOT kompiliert Ihre Anwendung in eine einzige native Binärdatei. Der Einstiegspunkt dieser Binärdatei ist die `static Main`-Methode. Innerhalb von `static Main` wird die Lambda-Laufzeit gebootstrapped und die `FunctionHandler`-Methode eingerichtet. Als Teil des Runtime-Bootstrap wird ein quellgenerierter Serializer mit `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()` konfiguriert

2. Um Ihre Anwendung auf Lambda bereitzustellen, stellen Sie sicher, dass Docker in Ihrer lokalen Umgebung ausgeführt wird, und führen Sie den folgenden Befehl aus.

```
dotnet lambda deploy-function
```

Hinter den Kulissen lädt das .NET Global CLI ein AL2023-Docker-Image herunter und kompiliert Ihren Anwendungscode in einem laufenden Container. Die kompilierte Binärdatei wird zurück in Ihr lokales Dateisystem ausgegeben, bevor sie auf Lambda bereitgestellt wird.

3. Testen Sie Ihre Funktion, indem Sie den folgenden Befehl ausführen. Ersetzen Sie `<FUNCTION_NAME>` durch den Namen, den Sie für Ihre Funktion im Bereitstellungsassistenten gewählt haben.

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

Die Antwort der CLI enthält Leistungsdetails für den Kaltstart (Initialisierungsdauer) und die Gesamtlaufzeit für Ihren Funktionsaufruf.

4. Führen Sie den folgenden Befehl aus, um die AWS Ressourcen zu löschen, die Sie mit den vorherigen Schritten erstellt haben. Ersetzen Sie `<FUNCTION_NAME>` durch den Namen, den Sie für Ihre Funktion im Bereitstellungsassistenten gewählt haben. Indem Sie AWS Ressourcen löschen, die Sie nicht mehr verwenden, verhindern Sie, dass Ihnen AWS-Konto unnötige Gebühren in Rechnung gestellt werden.

```
dotnet lambda delete-function <FUNCTION_NAME>
```

Serialisierung

Um Funktionen mithilfe von nativem AOT für Lambda bereitzustellen, muss Ihr Funktionscode die quellgenerierte Serialisierung verwenden. Anstatt mithilfe der Laufzeitreflexion die Metadaten zu sammeln, die für den Zugriff auf Objekteigenschaften für die Serialisierung erforderlich sind, generieren Quellgeneratoren C#-Quelldateien, die beim Erstellen Ihrer Anwendung kompiliert werden. Um Ihren quellgenerierten Serializer korrekt zu konfigurieren, stellen Sie sicher, dass Sie alle Eingabe- und Ausgabeobjekte, die Ihre Funktion verwendet, sowie alle benutzerdefinierten Typen einbeziehen. Beispielsweise würde eine Lambda-Funktion, die Ereignisse von einer API-Gateway-REST-API empfängt und einen benutzerdefinierten `Product`-Typ zurückgibt, einen Serializer enthalten, der wie folgt definiert ist.

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

Trimmen

Natives AOT kürzt Ihren Anwendungscode als Teil der Kompilierung, um sicherzustellen, dass die Binärdatei so klein wie möglich ist. .NET 8 for Lambda bietet im Vergleich zu früheren Versionen von .NET eine verbesserte Trimmunterstützung. Support wurde zu den Lambda-Laufzeitbibliotheken, AWS .NET SDK, .NET Lambda Annotations und .NET 8 selbst hinzugefügt.

Diese Verbesserungen bieten zwar das Potenzial, Warnungen beim Trimmen während der Erstellung zu vermeiden, aber .NET ist nie vollständig trimmsicher. Das bedeutet, dass Teile von Bibliotheken, auf die Ihre Funktion angewiesen ist, im Rahmen des Kompilierungsschritts möglicherweise herausgeschnitten werden. Sie können dies verwalten, indem Sie `TrimmerRootAssemblies` als Teil Ihrer `.csproj`-Datei definieren, wie im folgenden Beispiel gezeigt.

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

Beachten Sie, dass bei einer Trim-Warnung das Hinzufügen der Klasse, die die Warnung erzeugt, zu `TrimmerRootAssembly` das Problem möglicherweise nicht behebt. Eine Trim-Warnung weist darauf hin, dass die Klasse versucht, auf eine andere Klasse zuzugreifen, die erst zur Laufzeit ermittelt werden kann. Um Laufzeitfehler zu vermeiden, fügen Sie diese zweite Klasse zu `TrimmerRootAssembly` hinzu.

Weitere Informationen zur Verwaltung von Trimmwarnungen finden Sie in der Microsoft.NET-Dokumentation unter Einführung in Trimmwarnungen.

Fehlerbehebung

Fehler: Die Cross-OS native Kompilierung wird nicht unterstützt.

> Ihre Version des globalen Amazon.Lambda.Tools-.NET-Core-Tools ist veraltet. Aktualisieren Sie auf die neueste Version und versuchen Sie es erneut.

Docker: das Image-Betriebssystem „Linux“ kann auf dieser Plattform nicht verwendet werden.

> Docker ist auf Ihrem System für die Verwendung von Windows-Containern konfiguriert. Wechseln Sie zu Linux-Containern, um die native AOT-Entwicklungsumgebung auszuführen.

Weitere Informationen zu häufigen Fehlern finden Sie im AWS NativeAOT for .NET-Repository unter. GitHub

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur C#-Ereignisfunktion

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler übergeben. Dieses Objekt stellt Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Eigenschaften

- `FunctionName` – Der Name der Lambda-Funktion.
- `FunctionVersion` – Die Version der Funktion.
- `InvokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `MemoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `AwsRequestId` – Der Bezeichner der Aufrufanforderung.
- `LogGroupName` – Protokollgruppe für die Funktion.
- `LogStreamName` – Der Protokollstrom für die Funktionsinstance.
- `RemainingTime` (`TimeSpan`) – Die Anzahl der Millisekunden, die vor der Zeitüberschreitung der Ausführung verbleiben.
- `Identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `ClientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
- `Logger` Das Logger-Objekt für die Funktion.

Sie können die Informationen im `ILambdaContext`-Objekt verwenden, um zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion auszugeben. Der folgende Code enthält ein Beispiel dafür, wie Sie einem strukturierten Logging-Framework Kontextinformationen hinzufügen können. In diesem Beispiel fügt die Funktion `AwsRequestId` zu den Protokollausgaben hinzu. Die Funktion verwendet auch die `RemainingTime`-Eigenschaft, um eine Aufgabe während des Fluges abzubrechen, wenn die Zeitüberschreitung der Lambda-Funktion bald erreicht ist.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;
```

```
public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        catch (Exception ex)
        {
            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(new { error = ex.Message })
            };
        }
        finally
        {
            cts.Cancel();
        }
    }
```

```
}
```

Protokollieren und Überwachen von C#-Lambda-Ereignisfunktionen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterLambda Event Function-Logs an CloudWatch Logs senden.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET
- Zusätzliche Protokollierungswerkzeuge und Bibliotheken
- Verwendung von Powertools für AWS Lambda (.NET) und AWS SAM für strukturiertes Logging
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Logs aus Ihrem Funktionscode auszugeben, können Sie das ILambdaLoggerOn-the-Context-Objekt, die Methoden der Console-Klasse oder eine beliebige Logging-Bibliothek verwenden, die in `stdout` oder schreibt`stderr`.

Die .NET-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.

- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET

Um Ihnen mehr Kontrolle darüber zu geben, wie die Protokolle Ihrer Funktionen erfasst, verarbeitet und genutzt werden, können Sie die folgenden Protokollierungsoptionen für unterstützte.NET-Laufzeiten konfigurieren:

- Protokollformat — wählen Sie zwischen Klartext und strukturiertem JSON-Format für die Logs Ihrer Funktion
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, an die Lambda sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Logs sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen”.

Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren .NET-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden des strukturierten JSON-Protokollformats mit .NET

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die Protokollausgabe ILambdaLoggerals strukturiertes JSON. Jedes JSON-Protokollobjekt enthält mindestens fünf Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde

- `"level"` – die der Meldung zugewiesene Protokollebene
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf
- `"traceId"` – Die Umgebungsvariable `_X_AMZN_TRACE_ID`
- `"message"` – der Inhalt der Protokollmeldung

Die `ILambdaLogger`-Instance kann zusätzliche Schlüssel-Wert-Paare hinzufügen, beispielsweise beim Protokollieren von Ausnahmen. Sie können auch Ihre eigenen zusätzlichen Parameter angeben, wie im Abschnitt the section called “Customer-provided Log-Parameter” beschrieben.

> **Note**
>
> Wenn Ihr Code bereits eine andere Protokollierungsbibliothek zur Erstellung von JSON-formatted Protokollen verwendet, stellen Sie sicher, dass das Protokollformat Ihrer Funktion auf Klartext eingestellt ist. Wenn Sie das Protokollformat auf JSON setzen, werden Ihre Protokollausgaben doppelt codiert.

Der folgende Beispiel-Logging-Befehl zeigt, wie Sie eine Protokollnachricht mit dem Level `INFO` schreiben.

Example.NET-Protokollierungscode

```
context.Logger.LogInformation("Fetching cart from database");
```

Sie können auch eine generische Protokollebene verwenden, die die Protokollebene als Argument verwendet, wie im folgenden Beispiel gezeigt.

```
context.Logger.Log(LogLevel.Information, "Fetching cart from database");
```

Die Protokollausgabe dieser Beispielcodefragmente würde wie folgt in CloudWatch Logs erfasst:

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
```

```
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Fetching cart from database"
}
```

> **Note**
>
> Wenn Sie das Protokollformat Ihrer Funktion so konfigurieren, dass statt JSON reiner Text verwendet wird, entspricht die in der Meldung erfasste Protokollstufe der Microsoft-Konvention, die eine vierstellige Bezeichnung vorsieht. Die Protokollebene von `Debug` wird in der Nachricht beispielsweise als `dbug` dargestellt.
> Wenn Sie Ihre Funktion für die Verwendung von Protokollen im JSON-Format konfigurieren, verwendet die im Protokoll erfasste Protokollebene die vollständige Bezeichnung, wie im Beispiel für den JSON-Protokolldatensatz gezeigt.

Wenn Sie Ihrer Protokollausgabe kein Level zuweisen, weist Lambda ihr automatisch das Level INFO zu.

Protokollierungsausnahmen in JSON

Wenn Sie die strukturierte JSON-Protokollierung mit `ILambdaLogger` verwenden, können Sie Ausnahmen in Ihrem Code protokollieren, wie im folgenden Beispiel gezeigt.

Example Verwendung der Ausnahmeprotokollierung

```
try
{
    connection.ExecuteQuery(query);
}
catch(Exception e)
{
    context.Logger.LogWarning(e, "Error executing query");
}
```

Das von diesem Code ausgegebene Protokollformat wird im folgenden JSON-Beispiel gezeigt. Beachten Sie, dass die `message`-Eigenschaft im JSON mithilfe des im `LogWarning`-Aufruf angegebenen Nachrichtenarguments aufgefüllt wird, während die `errorMessage`-Eigenschaft aus der `Message`-Eigenschaft der Ausnahme selbst stammt.

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Warning",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Error executing query",
    "errorType": "System.Data.SqlClient.SqlException",
    "errorMessage": "Connection closed",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

Wenn das Protokollierungsformat Ihrer Funktion auf JSON gesetzt ist, gibt Lambda auch JSON-formatted Protokollmeldungen aus, wenn Ihr Code eine nicht abgefangene Ausnahme auslöst. Der folgende Beispielcodeausschnitt und die Protokollnachricht zeigen, wie nicht abgefangene Ausnahmen protokolliert werden.

Example Ausnahmecode

```
throw new ApplicationException("Invalid data");
```

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Error",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Invalid data",
    "errorType": "System.ApplicationException",
    "errorMessage": "Invalid data",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

Customer-provided Log-Parameter

Bei JSON-formatted Protokollnachrichten können Sie zusätzliche Protokollparameter angeben und diese in das Protokoll aufnehmen`message`. Der folgende Codeausschnitt zeigt einen Befehl zum Hinzufügen von zwei vom Benutzer angegebenen Parametern mit der Bezeichnung `retryAttempt`

und `uri`. Im Beispiel stammt der Wert dieser Parameter aus den Argumenten `retryAttempt` und `uriDestination`, die an den Protokollierungsbefehl übergeben wurden.

Example JSON-Protokollierungsbefehl mit zusätzlichen Parametern

```
context.Logger.LogInformation("Starting retry {retryAttempt} to make GET request to
 {uri}", retryAttempt, uriDestination);
```

Die von diesem Befehl ausgegebene Protokollmeldung ist im folgenden JSON-Beispiel dargestellt.

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Starting retry 1 to make GET request to http://example.com/",
    "retryAttempt": 1,
    "uri": "http://example.com/"
}
```

Tip

Sie können bei der Angabe zusätzlicher Parameter auch Positionseigenschaften anstelle von Namen verwenden. Beispielsweise könnte der Protokollierungsbefehl im vorherigen Beispiel auch wie folgt geschrieben werden:

```
context.Logger.LogInformation("Starting retry {0} to make GET request to {1}",
 retryAttempt, uriDestination);
```

Beachten Sie, dass Lambda bei der Angabe zusätzlicher Protokollierungsparameter diese als Top-Level-Eigenschaften im JSON-Protokollsatz erfasst. Dieser Ansatz unterscheidet sich von einigen gängigen .NET-Protokollierungsbibliotheken, wie z. B. `Serilog`, die zusätzliche Parameter in einem separaten untergeordneten Objekt erfassen.

Wenn das Argument, das Sie für einen zusätzlichen Parameter angeben, ein komplexes Objekt ist, verwendet Lambda standardmäßig die `ToString()`-Methode, um den Wert bereitzustellen. Um

anzugeben, dass ein Argument JSON-serialisiert werden soll, verwenden Sie das Präfix `@`, wie im folgenden Codeschnipsel gezeigt. In diesem Beispiel ist `User` ein Objekt mit den Eigenschaften `FirstName` und `LastName`.

Example JSON-Protokollierungsbefehl mit serialisiertem JSON-Objekt

```
context.Logger.LogInformation("User {@user} logged in", User);
```

Die von diesem Befehl ausgegebene Protokollmeldung ist im folgenden JSON-Beispiel dargestellt.

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "User {@user} logged in",
    "user":
    {
        "FirstName": "John",
        "LastName": "Doe"
    }
}
```

Wenn das Argument für einen zusätzlichen Parameter ein Array ist oder `IList` oder `IDictionary` implementiert, fügt Lambda das Argument der JSON-Protokollnachricht als Array hinzu, wie im folgenden Beispiel-JSON-Protokollsatz gezeigt. In diesem Beispiel nimmt `{users}` ein `IList`-Argument entgegen, das Instanzen der `User`-Eigenschaft mit demselben Format wie im vorherigen Beispiel enthält. Lambda wandelt dieses `IList` in ein Array um, wobei jeder Wert mit der `ToString`-Methode erstellt wird.

Example JSON-Protokolldatensatz mit einem **`IList`-Argument**

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{users} have joined the group",
    "users":
```

```
    [
        "Rosalez, Alejandro",
        "Stiles, John"
    ]
}
```

Sie können die Liste auch mit JSON serialisieren, indem Sie das @-Präfix in Ihrem Protokollierungsbefehl verwenden. Im folgenden JSON-Protokolldatensatz ist die Eigenschaft `users` in JSON serialisiert.

Example **`JSON-Protokolldatensatz mit einem serialisierten JSON-IList-Argument`**

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{@users} have joined the group",
    "users":
    [
        {
            "FirstName": "Alejandro",
            "LastName": "Rosalez"
        },
        {
            "FirstName": "John",
            "LastName": "Stiles"
        }
    ]
}
```

Verwenden der Filterung auf Protokollebene mit .NET

Durch die Konfiguration der Filterung auf Protokollebene können Sie festlegen, dass nur Protokolle mit einer bestimmten Detailebene oder niedriger an Logs gesendet werden. CloudWatch Informationen zur Konfiguration der Filterung auf Protokollebene für Ihre Funktion finden Sie unter the section called “Log-level filtern”.

AWS Lambda Um Ihre Protokollnachrichten nach Protokollebene zu filtern, können Sie entweder Protokolle im JSON-Format oder die `Console` .NET-Methoden zur Ausgabe von Protokollnachrichten verwenden. Um Protokolle im JSON-Format zu erstellen, konfigurieren Sie den Protokolltyp Ihrer Funktion auf JSON und verwenden Sie die `ILambdaLogger`-Instance.

Bei JSON-formatted Protokollen filtert Lambda Ihre Protokollausgaben mithilfe des Schlüsselwertpaars „level“ im JSON-Objekt, wie unter beschrieben. the section called “Verwenden des strukturierten JSON-Protokollformats mit .NET”

Wenn Sie die `Console` .NET-Methoden verwenden, um Nachrichten in CloudWatch Logs zu schreiben, wendet Lambda Log-Levels wie folgt auf Ihre Nachrichten an:

- Console.WriteLine method - Lambda wendet ein Log-Level von an `INFO`
- Console.Error method - Lambda wendet ein Log-Level von an `ERROR`

Wenn Sie Ihre Funktion für die Filterung auf Protokollebene konfigurieren, müssen Sie aus den folgenden Optionen für die Protokollebene auswählen, die Lambda an Logs senden soll. CloudWatch Beachten Sie die Zuordnung der von Lambda verwendeten Protokollebenen zu den Microsoft-Standardstufen, die von .NET `ILambdaLogger` verwendet werden.

Lambda-Protokollebene	Äquivalente Microsoft-Ebene	Standardnutzung
TRACE (am detailliertesten)	Trace	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Debuggen	Detaillierte Informationen für das System-Debugging
INFO	Informationen	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Warning	Meldungen über potenzielle Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Fehler	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert

Lambda-Protokollebene	Äquivalente Microsoft-Ebene	Standardnutzung
FATAL (am wenigsten Details)	Kritisch	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Lambda sendet Protokolle mit der ausgewählten Detailebene und niedriger bis CloudWatch. Wenn Sie beispielsweise die Protokollebene WARN konfigurieren, sendet Lambda Protokolle, die den Stufen WARN, ERROR und FATAL entsprechen.

Zusätzliche Protokollierungswerkzeuge und Bibliotheken

Powertools for AWS Lambda (.NET) ist ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit. Das Logging-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen zum Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)
- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu
- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (.NET) und AWS SAM für strukturiertes Logging

Gehen Sie wie folgt vor, um mithilfe von eine Hello World C#-Beispielanwendung mit integrierten Powertools for AWS Lambda (.NET) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-

API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET 8
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

Das Ergebnis sieht folgendermaßen aus:

```
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2025-09-20T14:15:27.988000 INIT_START Runtime Version:
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:29.259000
 2025-09-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.479000
 2025-09-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2025-09-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
```

```
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.599000
 2025-09-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2692 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
```

```
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Instrumentierung von C#-Code in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können es verwenden X-Ray, um eine Anfrage zu verfolgen, während sie Ressourcen in Ihrer Anwendung durchquert, zu denen Lambda-Funktionen und andere AWS Dienste gehören können.

Um Ablaufverfolgungsdaten an zu senden X-Ray, können Sie eine von drei SDK-Bibliotheken verwenden:

- AWS Distribution for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des (OTel) SDK. OpenTelemetry
- AWS X-Ray SDK für .NET— Ein SDK zum Generieren und Senden von Trace-Daten an. X-Ray
- Powertools for AWS Lambda (.NET) — Ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den Dienst zu senden. X-Ray Anschließend können Sie X-Ray die Leistungskennzahlen Ihrer Anwendung anzeigen, filtern und Einblicke in sie gewinnen, um Probleme und Optimierungsmöglichkeiten zu identifizieren.

> **Important**
>
> Die SDKs X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von angeboten wird. AWS Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgängige Ablaufverfolgung implementieren. X-Ray Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Auswahl zwischen der AWS Distribution for Open Telemetry und SDKs. X-Ray

Sections

- Verwenden von Powertools für AWS Lambda (.NET) und AWS SAM für die Rückverfolgung
- Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer .NET-Funktionen
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API

- Die Ablaufverfolgung wird aktiviert mit CloudFormation
- Interpretieren und X-Ray Trace

Verwenden von Powertools für AWS Lambda (.NET) und AWS SAM für die Rückverfolgung

Gehen Sie wie folgt vor, um eine Hello World C#-Beispielanwendung mit integrierten Powertools for AWS Lambda (.NET) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET 8
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie ein Beispiel bereit AWS SAM Anwendung

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

   > **Note**
   >
   > Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

   ```
   aws cloudformation describe-stacks --stack-name sam-app --query
    'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
   ```

6. Rufen Sie den API-Endpunkt auf:

   ```
   curl <URL_FROM_PREVIOUS_STEP>
   ```

   Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

   ```
   {"message":"hello world"}
   ```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

   ```
   sam traces
   ```

   Das Nachverfolgungsergebnis sieht folgendermaßen aus:

   ```
   New XRay Service Graph
     Start time: 2023-02-20 23:05:16+08:00
     End time: 2023-02-20 23:05:16+08:00
     Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
    [1]
       Summary_statistics:
         - total requests: 1
         - ok count(2XX): 1
         - error count(4XX): 0
         - fault count(5XX): 0
         - total response time: 2.814
     Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
    - Edges: []
   ```

```
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt

1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

Verwenden Sie das X-Ray SDK zur Instrumentierung Ihrer .NET-Funktionen

Sie können Ihren Funktionscode instrumentieren, um Metadaten aufzuzeichnen und Downstream-Aufrufe zu verfolgen. Um Details zu Aufrufen aufzuzeichnen, die Ihre Funktion an andere Ressourcen und Dienste vornimmt, verwenden Sie das AWS X-Ray SDK für .NET. Um das SDK zu erhalten, fügen Sie die `AWSXRayRecorder`-Pakete Ihrer Projektdatei hinzu.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

Es gibt eine Reihe von Nuget-Paketen, die automatische Instrumentierung für AWS SDKs, Entity Framework und HTTP-Anfragen bieten. Den vollständigen Satz der Konfigurationsoptionen finden Sie unter AWS X-Ray SDK for .NET im AWS X-Ray -Entwicklerhandbuch.

Nachdem Sie die gewünschten Nuget-Pakete hinzugefügt haben, konfigurieren Sie die automatische Instrumentierung. Es empfiehlt sich, diese Konfiguration außerhalb der Handler-Funktion Ihrer Funktion durchzuführen. Dabei wird die Wiederverwendung der Ausführungsumgebung genutzt, um die Leistung Ihrer Funktion zu verbessern. Im folgenden Codebeispiel wird die `RegisterXRayForAllServices` Methode im Funktionskonstruktor aufgerufen, um Instrumentierung für alle AWS SDK-Aufrufe hinzuzufügen.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer
```

```
namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.

5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpretieren und X-Ray Trace

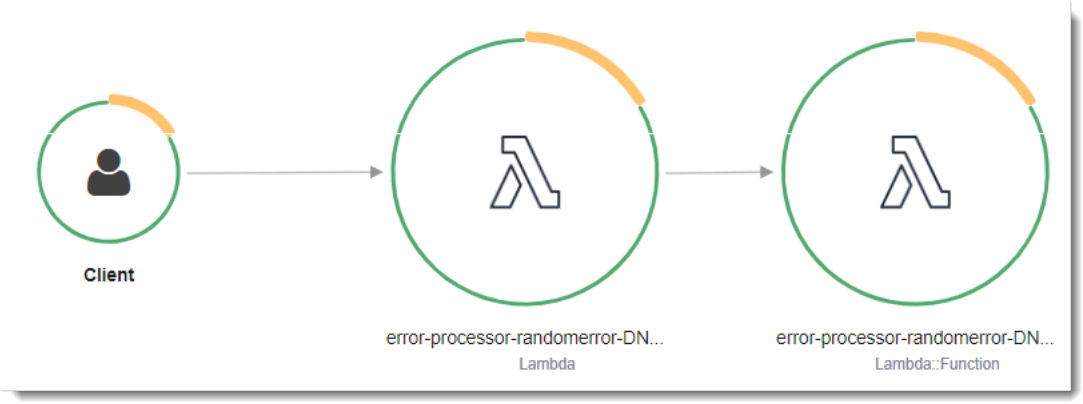
Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen X-Ray. Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray Dienstdiagramm zeigt Informationen zu Ihrer Anwendung und all ihren Komponenten. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK X-Ray, um Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

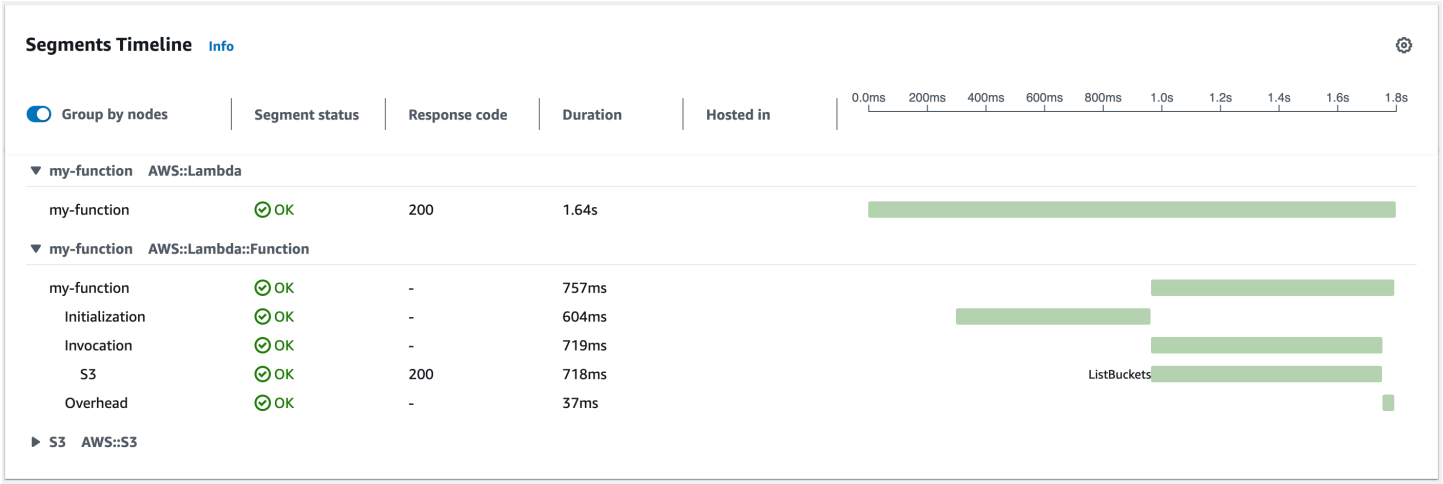


X-Ray verfolgt nicht alle Anfragen bis zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

In X-Ray zeichnet ein Trace Informationen über eine Anfrage auf, die von einem oder mehreren Diensten verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen stellen Sie möglicherweise geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollmeldungen und Trace-Segmenten fest, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “X-Ray Spuren verstehen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray SDK für .NET im AWS X-Ray -Entwicklerhandbuch.

Preisgestaltung

Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit kostenlos nutzen. AWS Bei Überschreitung dieses Schwellenwerts X-Ray fallen Gebühren für die Speicherung und den Abruf von Trace an. Weitere Informationen finden Sie unter AWS X-Ray Preise.

AWS Lambda Testen von Event-Funktionen in C#

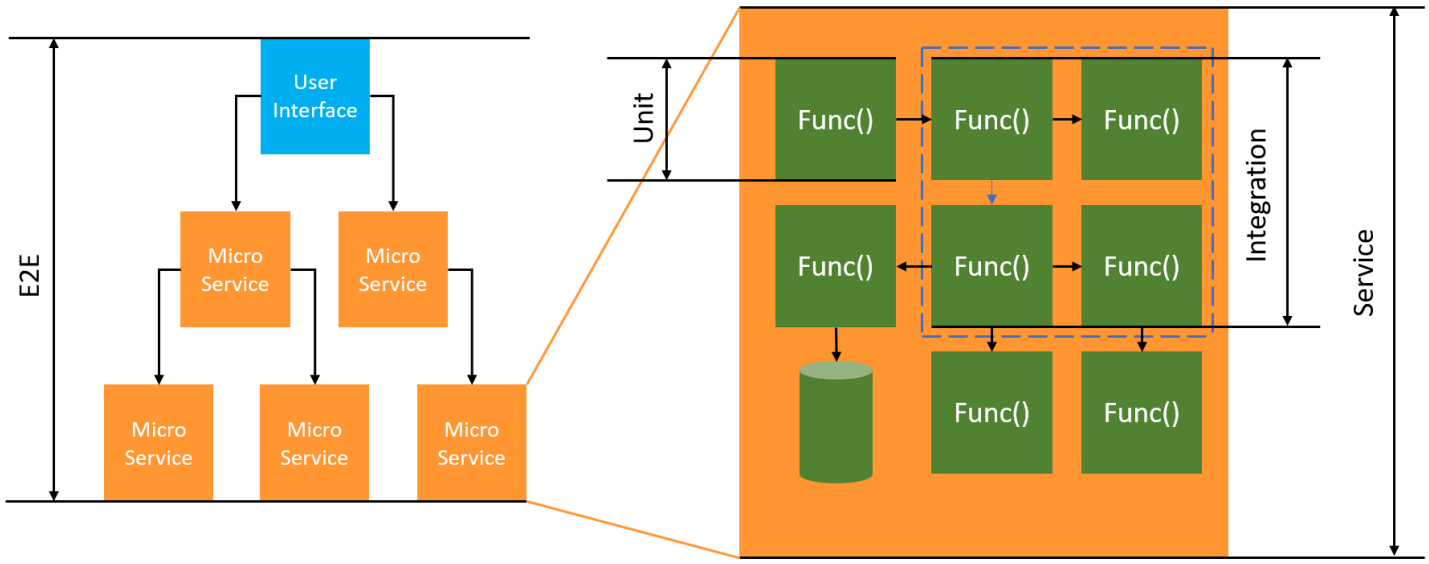
> **Note**
>
> Im Kapitel Testen von Funktionen finden Sie eine vollständige Einführung in Techniken und bewährte Methoden für das Testen von Serverless-Lösungen.

Beim Testen serverloser Funktionen werden herkömmliche Testtypen und -techniken verwendet, Sie müssen jedoch auch das Testen serverloser Anwendungen als Ganzes in Betracht ziehen. Cloud-based Tests bieten das genaueste Maß für die Qualität Ihrer Funktionen und serverlosen Anwendungen.

Eine Serverless-Anwendungsarchitektur umfasst verwaltete Services, die über API-Aufrufe wichtige Anwendungsfunktionen bereitstellen. Aus diesem Grund muss Ihr Entwicklungszyklus automatisierte Tests beinhalten, die bei der Interaktion Ihrer Funktionen und Services die Funktionalität überprüfen.

Wenn Sie keine cloud-basierten Tests erstellen, können aufgrund von Unterschieden zwischen Ihrer lokalen Umgebung und der bereitgestellten Umgebung Probleme auftreten. Ihr kontinuierlicher Integrationsprozess muss Tests anhand einer Reihe von Ressourcen durchführen, die in der Cloud bereitgestellt werden, bevor Ihr Code in die nächste Bereitstellungsumgebung wie QA, Staging oder Produktion übertragen wird.

Lesen Sie diesen kurzen Leitfaden weiter, um weitere Informationen zu Teststrategien für Serverless-Anwendungen zu erhalten, oder besuchen Sie das Serverless Test Samples Repository, um praktische Beispiele zu finden, die sich speziell auf die gewählte Sprache und Laufzeit beziehen.



Für serverlose Tests schreiben Sie immer noch Einheiten -, Integrations - und End-to-End-Tests.

- Einheitentests — Tests, die an einem isolierten Codeblock ausgeführt werden. Zum Beispiel die Überprüfung der Geschäftslogik zur Berechnung der Bereitstellungskosten für einen bestimmten Artikel und Bestimmungsort.
- Integrationstests — Tests, an denen zwei oder mehr Komponenten oder Dienste beteiligt sind, die interagieren, in der Regel in einer Cloud-Umgebung. Bei der Überprüfung einer Funktion werden beispielsweise Ereignisse aus einer Warteschlange verarbeitet.
- End-to-end Tests — Tests, die das Verhalten einer gesamten Anwendung überprüfen. Stellen Sie beispielsweise sicher, dass die Infrastruktur korrekt eingerichtet ist und die Ereignisse zwischen den Services wie erwartet ablaufen, um die Bestellungen der Kunden aufzuzeichnen.

Testen Ihrer Serverless-Anwendungen

Im Allgemeinen verwenden Sie verschiedene Methoden, um Ihren serverlosen Anwendungscode zu testen, darunter Tests in der Cloud, Tests mit Mocks und gelegentlich Tests mit Emulatoren.

Testen in der Cloud

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Sie führen Tests für Code durch, der in der Cloud bereitgestellt wird und mit cloud-basierten Services interagiert. Dieser Ansatz bietet das genaueste Maß für die Qualität Ihres Codes.

Eine bequeme Möglichkeit, Ihre Lambda Event Function in der Cloud zu debuggen, ist die Konsole mit einem Testereignis. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres JSON-Dokument (`{}`) sein. Die Konsole bietet Beispielereignisse für eine Vielzahl von Service-Integrationen. Nachdem Sie ein Ereignis in der Konsole erstellt haben, können Sie es mit Ihrem Team teilen, um das Testen einfacher und einheitlicher zu gestalten.

> **Note**
>
> Das Testen einer Funktion in der Konsole ist ein schneller Einstieg, aber die Automatisierung Ihrer Testzyklen gewährleistet die Anwendungsqualität und die Entwicklungsgeschwindigkeit.

Test-Tools

Um Ihren Entwicklungszyklus zu beschleunigen, gibt es eine Reihe von Tools und Techniken, die Sie beim Testen Ihrer Funktionen verwenden können. AWS SAM Accelerate und AWS CDK Watch Mode reduzieren beispielsweise beide die Zeit, die für die Aktualisierung von Cloud-Umgebungen benötigt wird.

Die Art und Weise, wie Sie Ihren Lambda-Funktionscode definieren, macht es einfach, Komponententests hinzuzufügen. Lambda benötigt einen öffentlichen, parameterlosen Konstruktor, um Ihre Klasse zu initialisieren. Durch die Einführung eines zweiten, internen Konstruktors haben Sie die Kontrolle über die Abhängigkeiten, die Ihre Anwendung verwendet.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
```

```
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Um einen Test für diese Funktion zu schreiben, können Sie eine neue Instance Ihrer `Function`-Klasse initialisieren und eine simulierte Implementierung von `IDatabaseRepository` übergeben. Die folgenden Beispiele verwenden `XUnit`, `Moq` und `FluentAssertions`, um einen einfachen Test zu schreiben, der sicherstellt, dass `FunctionHandler` einen 200-Statuscode zurückgibt.

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
```

```
    }
}
```

Ausführlichere Beispiele, einschließlich Beispiele für asynchrone Tests, finden Sie im Repository für .NET-Testbeispiele unter. GitHub

Lambda-Ereignisfunktionen erstellen mit PowerShell

In den folgenden Abschnitten wird erklärt, wie allgemeine Programmiermuster und Kernkonzepte beim Verfassen von Lambda-Ereignisfunktionscode in PowerShell angewendet werden.

Hinweise zum PowerShell Kontextobjekt finden Sie unterthe section called “Kontext”.

Lambda bietet die folgenden Beispielanwendungen für PowerShell:

- blank-powershell — Eine PowerShell Funktion, die die Verwendung von Logging, Umgebungsvariablen und dem SDK veranschaulicht. AWS

Bevor Sie beginnen, müssen Sie zunächst eine PowerShell Entwicklungsumgebung einrichten. Anweisungen dazu finden Sie unter Eine PowerShell Entwicklungsumgebung einrichten.

Weitere Informationen zur Verwendung des AWSLambdaPSCore Moduls zum Herunterladen von PowerShell Beispielprojekten aus Vorlagen, zum Erstellen von PowerShell Bereitstellungspaketen und zum Bereitstellen von PowerShell Funktionen in der AWS Cloud finden Sie unterStellen Sie PowerShell Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit.

Lambda stellt die folgenden Laufzeiten für .NET-Sprachen bereit:

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 10	`dotnet10`	Amazon Linux 2023	14. November 2028	14. Dezember 2028	15. Januar 2029
.NET 9 (nur Container)	`dotnet9`	Amazon Linux 2023	10. November 2026	Nicht geplant	Nicht geplant

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 8	`dotnet8`	Amazon Linux 2023	10. November 2026	29. Juli 2027	31. August 2027

Themen

- Eine PowerShell Entwicklungsumgebung einrichten
- Stellen Sie PowerShell Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit
- Definieren Sie den Lambda-Ereignisfunktionshandler in PowerShell
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur PowerShell Ereignisfunktion
- Powershell Lambda Event Functions protokollieren und überwachen

Eine PowerShell Entwicklungsumgebung einrichten

Lambda bietet eine Reihe von Tools und Bibliotheken für die PowerShell Laufzeit. Installationsanweisungen finden Sie unter Lambda-Tools für PowerShell on GitHub.

Das AWSLambdaPSCore Modul enthält die folgenden Cmdlets, die beim Verfassen und Veröffentlichen von PowerShell Lambda-Ereignisfunktionen helfen:

- Get-AWSPowerShellLambdaTemplate— Gibt eine Liste der Vorlagen für die ersten Schritte zurück.
- New-AWSPowerShellLambda— Erstellt ein erstes PowerShell Skript auf der Grundlage einer Vorlage.
- Publish-AWSPowerShellLambda— Veröffentlicht ein bestimmtes PowerShell Skript auf Lambda.
- New-AWSPowerShellLambdaPackage— Erstellt ein Lambda-Bereitstellungspaket, das Sie in einem CI/CD System für die Bereitstellung verwenden können.

Stellen Sie PowerShell Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit

Ein Bereitstellungspaket für die PowerShell Runtime enthält Ihr PowerShell Skript, PowerShell Module, die für Ihr PowerShell Skript erforderlich sind, und die Assemblys, die zum Hosten PowerShell von Core benötigt werden.

Die Lambda-Event-Funktion erstellen

Um mit dem Schreiben und Aufrufen eines PowerShell Skripts mit Lambda zu beginnen, können Sie mit dem `New-AWSPowerShellLambda` Cmdlet ein Startskript erstellen, das auf einer Vorlage basiert. Sie können das `Publish-AWSPowerShellLambda`-Cmdlet verwenden, um Ihr Skript in Lambda bereitzustellen. Anschließend können Sie Ihr Skript entweder über die Befehlszeile oder die Lambda-Konsole testen.

Gehen Sie wie folgt vor, um ein neues PowerShell Skript zu erstellen, es hochzuladen und zu testen:

1. Führen Sie den folgenden Befehl aus, um die Liste der verfügbaren Vorlagen anzuzeigen:

   ```
   PS C:\> Get-AWSPowerShellLambdaTemplate

   Template               Description
   --------               -----------
   Basic                  Bare bones script
   CodeCommitTrigger      Script to process AWS CodeCommit Triggers
   ...
   ```

2. Führen Sie den folgenden Befehl aus, um ein Beispielskript anhand der `Basic`-Vorlage zu erstellen:

   ```
   New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
   ```

   Eine neue Datei mit dem Namen `MyFirstPSScript.ps1` wird in einem neuen Unterverzeichnis des aktuellen Verzeichnisses erstellt. Der Name des Verzeichnisses basiert auf dem `-ScriptName`-Parameter. Sie können den `-Directory`-Parameter verwenden, um ein alternatives Verzeichnis auswählen.

   Sie können sehen, dass die neue Datei folgenden Inhalt hat:

   ```
   # PowerShell script file to run as a Lambda function
   #
   ```

```
# When executing in Lambda the following variables are predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
#
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. Um zu sehen, wie Protokollnachrichten von Ihrem PowerShell Skript an Amazon CloudWatch Logs gesendet werden, entfernen Sie das Kommentarzeichen aus der `Write-Host` Zeile des Beispielskripts.

   Um zu veranschaulichen, wie Sie Daten von Ihren Lambda-Funktionen zurückgeben können, fügen Sie mit `$PSVersionTable` eine neue Zeile am Ende des Skripts hinzu. Dadurch wird der `$PSVersionTable` PowerShell Pipeline hinzugefügt. Nach Abschluss des PowerShell Skripts sind das letzte Objekt in der PowerShell Pipeline die Rückgabedaten für die Lambda-Funktion. `$PSVersionTable`eist eine PowerShell globale Variable, die auch Informationen über die laufende Umgebung bereitstellt.

   Nachdem Sie alle Änderungen vorgenommen haben, sehen die beiden letzten Zeilen des Beispielskripts wie folgt aus:

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. Nachdem Sie die `MyFirstPSScript.ps1`-Datei bearbeitet haben, ändern Sie das Verzeichnis in den Speicherort des Skripts. Führen Sie anschließend führen Sie den folgenden Befehl aus, um das Skript in Lambda zu veröffentlichen:

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

Beachten Sie, dass der `-Name`-Parameter den Lambda-Funktionsnamen angibt, der in der Lambda-Konsole angezeigt wird. Sie können diese Funktion auch verwenden, um Ihr Skript manuell aufzurufen.

5. Rufen Sie Ihre Funktion mit dem `invoke` Befehl AWS Command Line Interface (AWS CLI) auf.

```
> aws lambda invoke --function-name MyFirstPSScript out
```

Definieren Sie den Lambda-Ereignisfunktionshandler in PowerShell

Wenn eine Lambda-Funktion aufgerufen wird, ruft der Lambda-Handler das Skript auf. PowerShell

Wenn das PowerShell Skript aufgerufen wird, sind die folgenden Variablen vordefiniert:

- *$LambdaInput*— Ein PSObject, das die Eingabe für den Handler enthält. Diese Eingabedaten können (von einer Ereignisquelle veröffentlichte) Ereignisdaten oder beliebige andere Eingabedaten sein, die Sie in Form einer Zeichenfolge oder eines benutzerdefinierten Datenobjekts übergeben.
- *$LambdaContext*— Ein Amazon.Lambda.Core.ILambdaContext Objekt, mit dem Sie auf Informationen über den aktuellen Aufruf zugreifen können, z. B. auf den Namen der aktuellen Funktion, das Speicherlimit, die verbleibende Ausführungszeit und die Protokollierung.

Betrachten Sie zum Beispiel den folgenden Beispielcode. PowerShell

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

Dieses Skript gibt die FunctionName Eigenschaft zurück, die aus der LambdaContext Variablen $ abgerufen wurde.

> **Note**
>
> Sie müssen die `#Requires` Anweisung in Ihren PowerShell Skripten verwenden, um anzugeben, von welchen Modulen Ihre Skripten abhängen. Diese Anweisung erfüllt zwei wichtige Aufgaben. 1) Es teilt anderen Entwicklern mit, welche Module das Skript verwendet, und 2) es identifiziert die abhängigen Module, die AWS PowerShell Tools im Rahmen der Bereitstellung mit dem Skript paketieren müssen. Weitere Informationen zu der `#Requires` Anweisung in finden Sie PowerShell unter About requires. Weitere Informationen zu PowerShell Bereitstellungspaketen finden Sie unterStellen Sie PowerShell Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit.
> Wenn Ihre PowerShell Lambda-Funktion die AWS PowerShell Cmdlets verwendet, stellen Sie sicher, dass Sie eine `#Requires` Anweisung festlegen, die auf das `AWSPowerShell.NetCore` Modul verweist, das PowerShell Core unterstützt, und nicht auf das `AWSPowerShell` Modul, das nur Windows unterstützt. PowerShell Stellen Sie zudem sicher, dass Sie `AWSPowerShell.NetCore` Version 3.3.270.0 oder höher verwenden, da

diese den Cmdlet-Importvorgang optimiert. Mit älteren Versionen dauern Kaltstarts länger. Weitere Informationen finden Sie unter AWS Tools for PowerShell.

Zurückgeben von Daten

Einige Lambda-Aufrufe sollen Daten zurück an ihren Aufrufer zurückgeben. Beispiel: Wenn ein Aufruf als Reaktion auf eine Webanforderung aus API Gateway erfolgt ist, muss unsere Lambda-Funktion die Antwort zurückgeben. Für PowerShell Lambda sind das letzte Objekt, das der PowerShell Pipeline hinzugefügt wurde, die Rückgabedaten des Lambda-Aufrufs. Wenn es sich bei dem Objekt um eine Zeichenfolge handelt, werden die Daten im Ist-Zustand zurückgegeben. Andernfalls wird das Objekt mithilfe des `ConvertTo-Json`-Cmdlets in JSON konvertiert.

Stellen Sie sich zum Beispiel die folgende PowerShell Anweisung vor, die die Pipeline erweitert: `$PSVersionTable` PowerShell

```
$PSVersionTable
```

Nachdem das PowerShell Skript abgeschlossen ist, sind die Rückgabedaten für die Lambda-Funktion das letzte Objekt in der PowerShell Pipeline. `$PSVersionTable`ist eine PowerShell globale Variable, die auch Informationen über die laufende Umgebung bereitstellt.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur PowerShell Ereignisfunktion

Wenn Lambda Ihre Funktion ausführt, werden Context-Informationen übergeben, indem eine `$LambdaContext`-Variable für den Handler verfügbar gemacht wird. Diese Variable stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Eigenschaften

- `FunctionName` – Der Name der Lambda-Funktion.
- `FunctionVersion` – Die Version der Funktion.
- `InvokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `MemoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `AwsRequestId` – Der Bezeichner der Aufrufanforderung.
- `LogGroupName` – Protokollgruppe für die Funktion.
- `LogStreamName` – Der Protokollstrom für die Funktionsinstance.
- `RemainingTime` – Die Anzahl der Millisekunden, die vor der Zeitüberschreitung der Ausführung verbleiben.
- `Identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `ClientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
- `Logger` – Das Logger-Objekt für die Funktion.

Der folgende PowerShell Codeausschnitt zeigt eine einfache Handler-Funktion, die einige der Kontextinformationen ausgibt.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

Powershell Lambda Event Functions protokollieren und überwachen

AWS Lambda überwacht Lambda Event Functions automatisch in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Funktion umfasst eine CloudWatch Logs-Protokollgruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Lambda Event Function-Logs an CloudWatch Logs senden.

Auf dieser Seite wird beschrieben, wie Sie eine Protokollausgabe aus dem Code Ihrer Lambda-Funktion erstellen und mithilfe der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Logs aus Ihrem Funktionscode auszugeben, können Sie Cmdlets on oder ein beliebiges Logging-Modul verwenden Microsoft.PowerShell.Utility, das in oder schreibt. `stdout` `stderr` Im folgenden Beispiel wird `Write-Host` verwendet.

Example function/Handler.ps1 — Protokollierung

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example Protokollformat

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 9867 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

Die .NET-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.

- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die Segment-ID. X-Ray
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe anzuzeigen, nachdem Sie eine Lambda-Ereignisfunktion aufgerufen haben.

Wenn Ihr Code im eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie die Testfunktion der Konsole verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Aufrufe der Lambda-Ereignisfunktion anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (/aws/lambda/***your-function-name***).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige

Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray zeichnet Details über die Anfrage und den Protokollstream im Trace auf.

Logs mit dem anzeigen AWS Command Line Interface (AWS CLI)

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Lambda-Eventfunktionen mit Rust erstellen

Da Rust zu nativem Code kompiliert wird, benötigen Sie keine spezielle Laufzeit, um Rust-Code als Lambda-Event-Funktion auszuführen. Verwenden Sie stattdessen den Rust-Runtime-Client, um Ihr Projekt lokal zu erstellen, und stellen Sie es dann mithilfe einer OS-only Runtime auf Lambda bereit. Wenn Sie eine OS-only Runtime verwenden, hält Lambda das Betriebssystem automatisch mit den neuesten Patches auf dem neuesten Stand. Hinweise zur Arbeit mit HTTP-Ereignissen finden Sie unterthe section called “HTTP-Ereignisse”. Informationen zur Verwendung von Ebenen mit Rust finden Sie unterthe section called “Layer”.

Tools und Bibliotheken für Rust

- AWS SDK für Rust: Das AWS SDK für Rust bietet Rust-APIs für die Interaktion mit den Infrastrukturdiensten von Amazon Web Services.
- Rust-Laufzeitclient für Lambda: Der Rust-Runtime-Client macht es einfach, in Rust geschriebene Lambda-Funktionen auszuführen.
- Cargo Lambda: Dies ist eine Open-Source-Erweiterung eines Drittanbieters für das Cargo-Befehlszeilentool, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht.

- Lambda HTTP: Diese Bibliothek bietet einen Wrapper für die Arbeit mit HTTP-Ereignissen.
- Lambda-Erweiterung: Diese Bibliothek bietet Unterstützung für das Schreiben von Lambda-Erweiterungen mit Rust.
- AWS Lambda Ereignisse: Diese Bibliothek bietet Typdefinitionen für gängige Integrationen mit Ereignisquellen.

Beispiele für Lambda-Anwendungen für Rust

- Grundlegende Lambda-Funktion: Eine Rust-Funktion, die zeigt, wie grundlegende Ereignisse verarbeitet werden.
- Lambda-Funktion mit Fehlerbehandlung: Eine Rust-Funktion, die zeigt, wie benutzerdefinierte Rust-Fehler in Lambda behandelt werden.
- Lambda-Funktion mit gemeinsam genutzten Ressourcen: Ein Rust-Projekt, das gemeinsam genutzte Ressourcen initialisiert, bevor die Lambda-Funktion erstellt wird.
- Lambda-HTTP-Ereignisse: Eine Rust-Funktion, die HTTP-Ereignisse verarbeitet.
- Lambda-HTTP-Ereignisse mit CORS-Headern: Eine Rust-Funktion, die Tower verwendet, um CORS-Header zu injizieren.
- Lambda-REST-API: Eine REST-API, die Axum und Diesel verwendet, um eine Verbindung zu einer PostgreSQL-Datenbank herzustellen.
- Serverlose Rust-Demo: Ein Rust-Projekt, das die Verwendung der Rust-Bibliotheken, der Protokollierung, der Umgebungsvariablen und des SDK von Lambda zeigt. AWS
- Basic Lambda-Erweiterung: Eine Rust-Erweiterung, die zeigt, wie grundlegende Erweiterungsereignisse verarbeitet werden.
- Lambda-Protokolle und Erweiterung für Amazon Data Firehose: Eine Rust-Erweiterung, die das Senden von Lambda-Protokollierung an Firehose aufzeigt.

Themen

- Definieren Sie Lambda-Ereignisfunktionshandler in Rust
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Rust-Ereignisfunktion
- Verarbeitung von HTTP-Ereignissen mit Rust
- Stellen Sie Rust Lambda Event Functions mit ZIP-Dateiarchiven bereit
- Arbeiten mit Ebenen für Rust Lambda Event Functions
- Protokollieren und überwachen Sie die Rust Lambda-Ereignisfunktionen

Definieren Sie Lambda-Ereignisfunktionshandler in Rust

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda Event Function-Handlern in Rust arbeiten, einschließlich Projektinitialisierung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Rust-Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einen Amazon Simple Storage Service (S3)-Bucket stellt. Weitere Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven”.

Themen

- Einrichten Ihres Rust-Handler-Projekts
- Beispiel für einen Rust Lambda Event Function Code
- Gültige Klassendefinitionen für Rust-Handler
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Verwendung der AWS SDK für Rust in deinem Handler
- Zugriff auf Umgebungsvariablen
- Geteilten Zustand verwenden
- Bewährte Codemethoden für Rust Lambda Event Functions

Einrichten Ihres Rust-Handler-Projekts

Wenn Sie mit Lambda-Ereignisfunktionen in Rust arbeiten, umfasst der Prozess das Schreiben Ihres Codes, das Kompilieren und das Bereitstellen der kompilierten Artefakte in Lambda. Der einfachste Weg, ein Lambda-Handler-Projekt in Rust einzurichten, ist die Verwendung von AWS Lambda Runtime for Rust. Trotz ihres Namens ist die AWS Lambda Runtime for Rust keine verwaltete Laufzeit im gleichen Sinne wie in Lambda für Python, Java oder Node.js. Stattdessen ist die AWS Lambda Runtime for Rust eine crate (`lambda_runtime`), die das Schreiben von Lambda-Funktionen in Rust und die Anbindung an die Ausführungsumgebung AWS Lambda unterstützt.

Verwenden Sie den folgenden Befehl, um Cargo Lambda zu installieren, eine Open-Source-Erweiterung eines Drittanbieters für das Cargo-Befehlszeilentool, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht:

```
cargo install cargo-lambda
```

Verwenden Sie nach der erfolgreichen Installation von `cargo-lambda` den folgenden Befehl, um ein neues Rust-Lambda-Funktions-Handler-Projekt zu initialisieren:

```
cargo lambda new example-rust
```

Wenn Sie diesen Befehl ausführen, stellt Ihnen die Befehlszeilenschnittstelle (CLI) einige Fragen zu Ihrer Lambda-Funktion:

- HTTP-Funktion — Wenn Sie beabsichtigen, Ihre Funktion über API Gateway oder eine Funktions-URL aufzurufen, antworten Sie mit Ja. Andernfalls antworten Sie mit Nein. Im Beispielcode auf dieser Seite rufen wir unsere Funktion mit einem benutzerdefinierten JSON-Ereignis auf und wählen daher Nein.
- Ereignistyp: Wenn Sie eine vordefinierte Ereignisform verwenden möchten, um Ihre Funktion aufzurufen, wählen Sie den richtigen erwarteten Ereignistyp aus. Andernfalls lassen Sie diese Option leer. Im Beispielcode auf dieser Seite rufen wir unsere Funktion mit einem benutzerdefinierten JSON-Ereignis auf, daher lassen wir diese Option leer.

Nachdem der Befehl erfolgreich ausgeführt wurde, geben Sie das Hauptverzeichnis Ihres Projekts ein:

```
cd example-rust
```

Mit diesem Befehl werden eine `generic_handler.rs`- und eine `main.rs`-Datei im `src`-Verzeichnis generiert. `generic_handler.rs` kann verwendet werden, um einen generischen Ereignis-Handler anzupassen. Die `main.rs`-Datei enthält Ihre Hauptanwendungslogik. Die `Cargo.toml`-Datei enthält Metadaten über Ihr Paket und listet dessen externe Abhängigkeiten auf.

Beispiel für einen Rust Lambda Event Function Code

Das folgende Beispiel für einen Rust-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`main.rs` Lambda-Ereignisfunktion**

```
use aws_sdk_s3::{Client, primitives::ByteStream};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};
use serde_json::Value;
use std::env;

#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}

async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;

    let bucket_name = env::var("RECEIPT_BUCKET")
        .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;

    let receipt_content = format!(
        "OrderID: {}\nAmount: ${:.2}\nItem: {}",
        order.order_id, order.amount, order.item
    );
    let key = format!("receipts/{}.txt", order.order_id);

    let config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
    let s3_client = Client::new(&config);

    upload_receipt_to_s3(&s3_client, &bucket_name, &key, &receipt_content).await?;

    Ok("Success".to_string())
}

async fn upload_receipt_to_s3(
    client: &Client,
    bucket_name: &str,
    key: &str,
    content: &str,
```

```
) -> Result<(), Error> {
    client
        .put_object()
        .bucket(bucket_name)
        .key(key)
        .body(ByteStream::from(content.as_bytes().to_vec()))  // Fixed conversion
        .content_type("text/plain")
        .send()
        .await?;

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Diese `main.rs`-Datei enthält die folgenden Abschnitte des Codes:

- `use`-Anweisungen: Damit können Sie Rust-Crates und -Methoden importieren, die für Ihre Lambda-Funktion erforderlich sind.
- `#[derive(Deserialize, Serialize)]`: Definieren Sie die Form des erwarteten Eingabeereignisses in dieser Rust-Struktur.
- `async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `async fn upload_receipt_to_s3 (...)`: Dies ist eine Hilfsmethode, auf die von der Haupt-`function_handler`-Methode verwiesen wird.
- `#[tokio::main]`: Dies ist ein Makro, das den Einstiegspunkt eines Rust-Programms markiert. Außerdem wird eine Tokio-Laufzeit eingerichtet, die es Ihrer `main()`-Methode ermöglicht, `async`/`await` asynchron zu verwenden und auszuführen.
- `async fn main() -> Result<(), Error>`: Die `main()`-Funktion ist der Einstiegspunkt Ihres Codes. Darin spezifizieren wir `function_handler` als Haupt-Handler-Methode.

Cargo.toml Beispieldatei

Die folgende `Cargo.toml`-Datei gehört zu dieser Funktion.

```
[package]
```

```
name = "example-rust"
version = "0.1.0"
edition = "2024"

[dependencies]
aws-config = "1.5.18"
aws-sdk-s3 = "1.78.0"
lambda_runtime = "0.13.0"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tokio = { version = "1", features = ["full"] }
```

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

Gültige Klassendefinitionen für Rust-Handler

In den meisten Fällen haben Lambda-Handler-Signaturen, die Sie in Rust definieren, das folgende Format:

```
async fn function_handler(event: LambdaEvent<T>) -> Result<U, Error>
```

Für diesen Handler:

- Der Name dieses Handlers ist `function_handler`.
- Der einzige Eingabewert für den Handler ist „event“ und hat den Typ `LambdaEvent<T>`.
  - `LambdaEvent` ist ein Wrapper, die aus der `lambda_runtime`-Crate stammt. Durch die Verwendung dieses Wrappers erhalten Sie Zugriff auf das Kontextobjekt, das Lambda-specific Metadaten wie die Anforderungs-ID des Aufrufs enthält.
  - `T` ist der deserialisierte Ereignistyp. Dies kann beispielsweise `serde_json::Value` sein, wodurch der Handler jede generische JSON-Eingabe entgegennehmen kann. Alternativ kann dies ein Typ wie `ApiGatewayProxyRequest` sein, wenn Ihre Funktion einen bestimmten, vordefinierten Eingabetyp erwartet.
- Der Rückgabetyp des Handlers ist `Result<U, Error>`.
  - `U` ist der deserialisierte Ausgabetyp. `U` muss das `serde::Serialize`-Merkmal implementieren, damit Lambda den Rückgabewert in JSON konvertieren kann. Beispielsweise kann `U` ein

einfacher Typ wie `String`, `serde_json::Value` oder eine benutzerdefinierte Struktur sein, solange es `Serialize` implementiert. Wenn Ihr Code eine Ok(U)-Anweisung erreicht, bedeutet dies, dass die Ausführung erfolgreich war, und Ihre Funktion gibt einen Wert vom Typ U zurück.

- Wenn Ihr Code auf einen Fehler stößt (d. h.`Err(Error)`), protokolliert Ihre Funktion den Fehler in Amazon CloudWatch und gibt eine Fehlerantwort vom Typ zurück`Error`.

In unserem Beispiel sieht die Handler-Signatur wie folgt aus:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>
```

Andere gültige Handler-Signaturen können Folgendes enthalten:

- Weglassen des `LambdaEvent`-Wrappers: Wenn Sie `LambdaEvent` weglassen, verlieren Sie den Zugriff auf das Lambda-Kontextobjekt innerhalb Ihrer Funktion. Es folgt ein Beispiel für diese Art von Signatur:

  ```
  async fn handler(event: serde_json::Value) -> Result<String, Error>
  ```

- Verwendung des Einheitentyps als Eingabe: Bei Rust können Sie den Einheitentyp verwenden, um eine leere Eingabe darzustellen. Dies wird häufig für Funktionen mit periodischen, geplanten Aufrufen verwendet. Es folgt ein Beispiel für diese Art von Signatur:

  ```
  async fn handler(_: ()) -> Result<Value, Error>
  ```

Namenskonventionen für Handler

Lambda-Handler in Rust haben keine strengen Namensbeschränkungen. Sie können zwar einen beliebigen Namen für Ihren Handler verwenden, Funktionsnamen in Rust werden aber im Allgemeinen in `snake_case` angegeben.

Für kleinere Anwendungen wie in diesem Beispiel können Sie eine einzige `main.rs`-Datei verwenden, die Ihren gesamten Code enthält. Bei größeren Projekten sollte `main.rs` den Einstiegspunkt zu Ihrer Funktion enthalten, aber Sie können zusätzliche Dateien dafür haben, die Ihren Code in logische Module unterteilen. Beispielsweise könnte Ihre Dateistruktur wie folgt aussehen:

```
/example-rust
### src/
```

```
#   ### main.rs        # Entry point
#   ### handler.rs     # Contains main handler
#   ### services.rs    # [Optional] Back-end service calls
#   ### models.rs      # [Optional] Data models
### Cargo.toml
```

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gängigste und standardmäßigste Eingabeformat für Lambda Event Functions. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

In Rust können Sie die Form des erwarteten Eingabeereignisses in einer Struktur definieren. In diesem Beispiel definieren wir die folgende Struktur, die einen `Order` darstellt:

```
#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}
```

Diese Struktur entspricht der erwarteten Eingabeform. In diesem Beispiel generiert das `#[derive(Deserialize, Serialize)]`-Makro automatisch Code für die Serialisierung und Deserialisierung. Das bedeutet, dass wir den generischen Eingabe-JSON-Typ mithilfe der `serde_json::from_value()`-Methode in unsere Struktur deserialisieren können. Dies wird in den ersten Zeilen des Handlers veranschaulicht:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;
    ...
}
```

Sie können dann auf die Felder des Objekts zugreifen. Zum Beispiel holt `order.order_id` den Wert von `order_id` aus der ursprünglichen Eingabe.

Pre-defined Ereignistypen eingeben

In der `aws_lambda_events`-Crate sind viele vordefinierte Eingabe-Ereignistypen verfügbar. Wenn Sie beispielsweise beabsichtigen, Ihre Funktion mit API Gateway aufzurufen, einschließlich des folgenden Imports:

```
use aws_lambda_events::event::apigw::ApiGatewayProxyRequest;
```

Stellen Sie dann sicher, dass Ihr Haupt-Handler die folgende Signatur verwendet:

```
async fn handler(event: LambdaEvent<ApiGatewayProxyRequest>) -> Result<String, Error> {
    let body = event.payload.body.unwrap_or_default();
    ...
}
```

Weitere Informationen zu anderen vordefinierten Eingabe-Ereignistypen finden Sie in der Crate aws_lambda_events.

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In Rust enthält der `LambdaEvent`-Wrapper das Kontextobjekt. Beispielsweise können Sie das Kontextobjekt verwenden, um die Anforderungs-ID des aktuellen Aufrufs mit dem folgenden Code abzurufen:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let request_id = event.context.request_id;
    ...
}
```

Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Kontext”.

Verwendung der AWS SDK für Rust in deinem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für Rust.

Um SDK-Abhängigkeiten zu Ihrer Funktion hinzuzufügen, fügen Sie diese in Ihrer `Cargo.toml`-Datei hinzu. Wir empfehlen, nur die Bibliotheken hinzuzufügen, die Sie für Ihre Funktion benötigen. Im Beispielcode zuvor haben wir `aws_sdk_s3::Client` verwendet. In der `Cargo.toml`-Datei können Sie diese Abhängigkeit hinzufügen, indem Sie die folgende Zeile unter dem `[dependencies]`-Abschnitt hinzufügen:

```
aws-sdk-s3 = "1.78.0"
```

> **Note**
>
> Dies ist möglicherweise nicht die neueste Version. Wählen Sie die passende Version für Ihre Anwendung.

Importieren Sie dann die Abhängigkeiten direkt in Ihren Code:

```
use aws_sdk_s3::{Client, primitives::ByteStream};
```

Der Beispielcode initialisiert dann einen Amazon-S3-Client wie folgt:

```
let config = aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let s3_client = Client::new(&config);
```

Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS Diensten verwenden. Der Beispielcode ruft die Amazon-S3-`PutObject`-API in der Hilfsfunktion `upload_receipt_to_s3` auf.

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `env::var`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
let bucket_name = env::var("RECEIPT_BUCKET")
    .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;
```

Geteilten Zustand verwenden

Sie können geteilte Variablen deklarieren, die unabhängig vom Handler-Code Ihrer Lambda-Funktion sind. Diese Variablen können Ihnen helfen, Zustandsinformationen während des Init-Phase zu laden, bevor Ihre Funktion Ereignisse empfängt. Beispielsweise können Sie den Code auf dieser Seite so ändern, dass bei der Initialisierung des Amazon-S3-Clients ein gemeinsamer Status verwendet wird, indem Sie die `main`-Funktion und die Handler-Signatur aktualisieren:

```
async fn function_handler(client: &Client, event: LambdaEvent<Value>) -> Result<String,
 Error> {
    ...
    upload_receipt_to_s3(client, &bucket_name, &key, &receipt_content).await?;
    ...
}

...

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
```

Bewährte Codemethoden für Rust Lambda Event Functions

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die Best Practices für Codierung anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Dies macht es einfacher, eine Funktion zu erstellen, die besser anhand von Einheiten getestet werden kann.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch wird die Zeit reduziert, die benötigt wird, bis Ihr Bereitstellungspaket vor dem Aufruf heruntergeladen und entpackt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit in wiederverwenden. Keep-Alive Node.js

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates auf die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte

Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Informationen zur Rust-Ereignisfunktion

Wenn Lambda Ihre Event-Funktion ausführt, fügt es dem, das der Handler empfängt LambdaEvent , ein Kontextobjekt hinzu. Dieses Objekt stellt Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Eigenschaften

- `request_id`: Die vom Lambda-Service generierte AWS Anforderungs-ID.
- `deadline`: Die Ausführungsfrist für den aktuellen Aufruf in Millisekunden.
- `invoked_function_arn`: Der Amazon-Ressourcenname (ARN) der aufgerufenen Lambda-Funktion.
- `xray_trace_id`: Die AWS X-Ray Trace-ID für den aktuellen Aufruf.
- `client_content`: Das vom SDK für AWS Mobilgeräte gesendete Client-Kontextobjekt. Dieses Feld ist leer, es sei denn, die Funktion wird mit einem AWS mobilen SDK aufgerufen.
- `identity`: Die Amazon-Cognito-Identität, die die Funktion aufgerufen hat. Dieses Feld ist leer, es sei denn, die Aufrufanforderung an die Lambda-APIs erfolgte unter Verwendung von AWS -Anmeldeinformationen, die von Amazon-Cognito-Identitätspools ausgestellt wurden.
- `env_config`: Die Lambda-Funktionskonfiguration aus den lokalen Umgebungsvariablen. Diese Eigenschaft umfasst Informationen wie den Funktionsnamen, die Speicherzuweisung, die Version und die Protokollstreams.

Zugreifen auf Aufrufkontextinformationen

Lambda-Funktionen haben Zugriff auf Metadaten über ihre Umgebung und die Aufrufanforderung. Das `LambaEvent` Objekt, das Ihr Funktionshandler empfängt, enthält die `context`-Metadaten:

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}

#[tokio::main]
```

```
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Verarbeitung von HTTP-Ereignissen mit Rust

Amazon API Gateway APIs, Application Load Balancer und Lambda-Funktions-URLs können HTTP-Ereignisse an Lambda senden. Sie können den aws_lambda_events Crate von crates.io verwenden, um Ereignisse aus diesen Quellen zu verarbeiten.

Example– API Gateway Proxy-Anfrage bearbeiten

Beachten Sie Folgendes:

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`: Der aws_lambda_events Crate enthält viele Lambda-Ereignisse. Um die Kompilierungszeit zu verkürzen, verwenden Sie Feature-Flags, um die benötigten Ereignisse zu aktivieren. Beispiel: `aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`.
- `use http::HeaderMap`: Für diesen Import müssen Sie den HTTP Crate zu Ihren Abhängigkeiten hinzufügen.

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
```

```
}
```

Der Rust-Laufzeitclient für Lambda bietet auch eine Abstraktion über diese Ereignistypen, sodass Sie mit nativen HTTP-Typen arbeiten können, unabhängig davon, welcher Dienst die Ereignisse sendet. Der folgende Code entspricht dem vorherigen Beispiel und funktioniert sofort mit Lambda-Funktions-URLs, Application Load Balancer und API Gateway.

> **Note**
>
> Der lambda_http Crate verwendet den darunter liegenden lambda_runtime Crate. Sie müssen `lambda_runtime` nicht separat importieren.

Example– Bearbeitung von HTTP-Anforderungen

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

Ein weiteres Beispiel für die Verwendung `lambda_http` finden Sie im Codebeispiel http-axum im Labs-Repository. AWS GitHub

Beispiel für HTTP-Lambda-Ereignisse für Rust

- Lambda-HTTP-Ereignisse: Eine Rust-Funktion, die HTTP-Ereignisse verarbeitet.
- Lambda-HTTP-Ereignisse mit CORS-Headern: Eine Rust-Funktion, die Tower verwendet, um CORS-Header zu injizieren.

- Lambda-HTTP-Ereignisse mit gemeinsam genutzten Ressourcen: Eine Rust-Funktion, die gemeinsam genutzte Ressourcen verwendet, die initialisiert werden, bevor der Funktionshandler erstellt wird.

Stellen Sie Rust Lambda Event Functions mit ZIP-Dateiarchiven bereit

Diese Seite beschreibt, wie Sie Ihre Rust-Event-Funktion kompilieren und dann die kompilierte Binärdatei bereitstellen AWS Lambda unter Verwendung von Cargo Lambda. Es zeigt auch, wie man die kompilierte Binärdatei mit dem AWS Command Line Interface und der AWS Serverless Application Model CLIP.

Sections

- Voraussetzungen
- Aufbau von Rust-Funktionen auf macOS, Windows oder Linux
- Bereitstellung der Binärdatei der Rust-Funktion mit Cargo Lambda
- Aufrufen Ihrer Rust-Funktion mit Cargo Lambda

Voraussetzungen

- Rust
- AWS CLI Ausführung 2

Aufbau von Rust-Funktionen auf macOS, Windows oder Linux

Die folgenden Schritte zeigen, wie Sie das Projekt für Ihre erste Lambda-Funktion mit Rust erstellen und mit kompilieren Lambda transportieren, eine Open-Source-Erweiterung des Cargo-Befehlszeilentools von Drittanbietern, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht.

1. Installieren Cargo Lambda, eine Open-Source-Erweiterung des Cargo-Befehlszeilentools von Drittanbietern, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht:

   ```
   cargo install cargo-lambda
   ```

   Weitere Installationsoptionen finden Sie unter Installation in der Cargo Lambda-Dokumentation.

2. Erstellen Sie die Paketstruktur. Dieser Befehl erstellt einen grundlegenden Funktionscode in `src/main.rs`. Sie können diesen Code zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

   ```
   cargo lambda new my-function
   ```

3. Führen Sie im Stammverzeichnis des Pakets den Unterbefehl build aus, um den Code in Ihrer Funktion zu kompilieren.

```
cargo lambda build --release
```

(Optional) Wenn Sie verwenden möchten AWS Graviton auf Lambda, füge das hinzu `--arm64` Flag, um Ihren Code für ARM-CPUs zu kompilieren.

```
cargo lambda build --release --arm64
```

4. Bevor Sie Ihre Rust-Funktion bereitstellen, konfigurieren Sie AWS Anmeldeinformationen auf Ihrem Computer.

```
aws configure
```

Bereitstellung der Binärdatei der Rust-Funktion mit Cargo Lambda

Verwenden Sie den Unterbefehl bereitstellen, um die kompilierte Binärdatei für Lambda bereitzustellen. Dieser Befehl erstellt eine Ausführungsrolle und erstellt dann die Lambda-Funktion. Um eine bestehende Ausführungsrolle anzugeben, verwenden Sie die --iam-role Flag.

```
cargo lambda deploy my-function
```

Bereitstellen Ihrer Rust-Funktionsbinärdatei mit dem AWS CLI

Sie können Ihre Binärdatei auch mit dem bereitstellen AWS CLI.

1. Verwenden Sie den Unterbefehl build, um das .zip-Bereitstellungspaket zu erstellen.

```
cargo lambda build --release --output-format zip
```

2. Um das .zip-Paket in Lambda bereitzustellen, führen Sie den Befehl create-function aus.

   - Legen Sie für `--runtime` die Option `provided.al2023` fest. Das ist ein OS-only Laufzeit. OS-only Laufzeiten werden verwendet, um kompilierte Binärdateien und benutzerdefinierte Laufzeiten auf Lambda bereitzustellen.
   - Für `--role` geben Sie den ARN der Ausführungsrolle an.

```
aws lambda create-function \
     --function-name my-function \
     --runtime provided.al2023 \
     --role arn:aws:iam::111122223333:role/lambda-role \
     --handler rust.handler \
     --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

Bereitstellen Ihrer Rust-Funktionsbinärdatei mit dem AWS SAM CLI

Sie können Ihre Binärdatei auch mit dem bereitstellen AWS SAM CLI.

1. Erstellen Sie eine AWS SAM Vorlage mit der Ressourcen- und Eigenschaftsdefinition. Legen Sie für `Runtime` die Option `provided.al2023` fest. Das ist ein OS-only Laufzeit. OS-only Laufzeiten werden verwendet, um kompilierte Binärdateien und benutzerdefinierte Laufzeiten auf Lambda bereitzustellen.

   Weitere Informationen zur Bereitstellung von Lambda-Funktionen finden Sie unter AWS SAM, siehe AWS::Serverless::Funktion in der AWS Serverless Application Model Leitfaden für Entwickler.

   Example SAM-Ressourcen- und Eigenschaftsdefinition für eine Rust-Binärdatei

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: AWS::Serverless-2016-10-31
   Description: SAM template for Rust binaries
   Resources:
     RustFunction:
       Type: AWS::Serverless::Function
       Properties:
         CodeUri: target/lambda/my-function/
         Handler: rust.handler
         Runtime: provided.al2023
   Outputs:
     RustFunction:
       Description: "Lambda Function ARN"
       Value: !GetAtt RustFunction.Arn
   ```

2. Verwenden Sie den Unterbefehl build, um die Funktion zu kompilieren.

```
cargo lambda build --release
```

3. Verwenden Sie den Befehl sam deploy, um die Funktion in Lambda bereitzustellen.

```
sam deploy --guided
```

Weitere Informationen zum Erstellen von Rust-Funktionen finden Sie mit dem AWS SAM CLI, siehe Erstellen von Rust Lambda-Funktionen mit Cargo Lambda in der AWS Serverless Application Model Leitfaden für Entwickler.

Aufrufen Ihrer Rust-Funktion mit Cargo Lambda

Verwenden Sie den Unterbefehl invoke, um Ihre Funktion mit einer Nutzlast zu testen.

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

Aufrufen Ihrer Rust-Funktion mit dem AWS CLI

Sie können auch die verwenden AWS CLI um die Funktion aufzurufen.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

Das cli-binary-format Die Option ist erforderlich, wenn Sie verwenden AWS CLI Version 2. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Arbeiten mit Ebenen für Rust Lambda Event Functions

Wir empfehlen nicht, Ebenen zu verwenden, um Abhängigkeiten für in Rust geschriebene Lambda-Ereignisfunktionen zu verwalten. Dies liegt daran, dass Rust-Funktionen zu einer einzigen ausführbaren Datei kompiliert werden, die Sie Lambda bei der Bereitstellung Ihrer Funktion zur Verfügung stellen. Diese ausführbare Datei enthält Ihren kompilierten Funktionscode zusammen mit all seinen Abhängigkeiten. Die Verwendung von Ebenen verkompliziert nicht nur diesen Vorgang, sondern führt auch zu längeren Kaltstartzeiten, da Ihre Funktionen während der Init-Phase zusätzliche Assemblys manuell in den Speicher laden müssen.

Um externe Abhängigkeiten mit Ihren Rust-Handlern zu verwenden, fügen Sie diese direkt in Ihr Bereitstellungspaket ein. So vereinfachen Sie den Bereitstellungsprozess und profitieren außerdem von den integrierten Optimierungen des Rust-Compilers. Ein Beispiel dafür, wie Sie eine Abhängigkeit wie das AWS SDK für Rust in Ihre Funktion importieren und verwenden, finden Sie unter the section called “Handler”.

Protokollieren und überwachen Sie die Rust Lambda-Ereignisfunktionen

AWS Lambda überwacht Lambda Event Functions automatisch in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter.

Weitere Informationen finden Sie unter Lambda Event Function-Logs an CloudWatch Logs senden. Hinweise zur Konfiguration von Protokollformaten finden Sie unterKonfiguration der JSON- und Klartext-Protokollformate. Auf dieser Seite wird beschrieben, wie Sie Protokollausgaben aus dem Code Ihrer Lambda-Funktion erstellen.

Erstellen einer Funktion, die Protokolle schreibt

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie jede Protokollierungsfunktion verwenden, die in `stdout` oder `stderr` schreibt, wie z. B. das `println!`-Makro. Das folgende Beispiel verwendet `println!`, um eine Nachricht zu drucken, wenn der Funktionshandler gestartet wird und bevor er beendet wird.

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Implementierung der erweiterten Protokollierung mit der Tracing Crate

Tracing ist ein Framework zur Instrumentierung von Rust-Programmen zur Erfassung strukturierter, ereignisbasierter Diagnoseinformationen. Dieses Framework bietet Hilfsprogramme zum Anpassen der Logging-Ausgabestufen und -formate, wie zum Beispiel das Erstellen strukturierter JSON-Protokollnachrichten. Um dieses Framework verwenden zu können, müssen Sie einen `subscriber`

initialisieren, bevor Sie den Funktionshandler implementieren. Anschließend können Sie Tracing-Makros wie `debug`, `info` und `error` verwenden, um die gewünschte Protokollierungsebene für jedes Szenario anzugeben.

Example– Verwenden von Tracing Crate

Beachten Sie Folgendes:

- `tracing_subscriber::fmt().json()`: Wenn diese Option enthalten ist, werden die Protokolle in JSON formatiert. Um diese Option verwenden zu können, müssen Sie das `json`-Feature in die `tracing-subscriber`-Abhängigkeit aufnehmen (z. B. `tracing-subscriber = { version = "0.3.11", features = ["json"] }`).
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`: Diese Annotation generiert bei jedem Aufruf des Handlers eine Spanne. Diese Spanne fügt jeder Protokollzeile die Anforderungs-ID hinzu.
- `{ %first_name }`: Dieses Konstrukt fügt das `first_name`-Feld der Protokollzeile hinzu, in der es verwendet wird. Der Wert für dieses Feld entspricht der Variablen mit dem gleichen Namen.

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
```

```
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
}
```

Wenn diese Rust-Funktion aufgerufen wird, druckt sie zwei Protokollzeilen, die den folgenden ähneln:

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

Laufzeiten der Lambda-Event-Funktion

AWS Lambda Event-Funktionen (früher bekannt als Lambda-Funktionen) unterstützen mehrere Sprachen mithilfe von Laufzeiten. Eine Laufzeit bietet eine sprachspezifische Umgebung, die Aufrufereignisse, Kontextinformationen und Antworten zwischen Lambda und der Funktion weiterleitet. Sie können von Lambda bereitgestellte Laufzeiten verwenden oder Ihre eigenen erstellen. Hinweise zu Open-Source-Laufzeiten finden Sie unter the section called “Open-Source-Repositorys”. Informationen zu Laufzeiten, die Lambda Web Functions verwenden, finden Sie unter the section called “Laufzeiten”.

Lambda ist unabhängig von Ihrer Wahl der Laufzeit. Für einfache Funktionen, interpretierte Sprachen wie Python und Node.js bieten die schnellste Leistung. Bei Funktionen mit komplexeren Berechnungen lassen sich kompilierte Sprachen wie Java oft langsamer initialisieren, werden aber im Lambda-Handler schnell ausgeführt. Die Wahl der Laufzeit hängt auch von der Präferenz der Entwickler und der Vertrautheit mit der Sprache ab.

Jede größere Programmiersprachenversion verfügt über eine separate Laufzeit mit einer eindeutigen Laufzeitkennung, z. B. `nodejs24.x` oder `python3.14`. Um eine Funktion für die Verwendung einer neuen Hauptsprachversion zu konfigurieren, müssen Sie die Laufzeitkennung ändern. Seit AWS Lambda kann die Abwärtskompatibilität zwischen Hauptversionen nicht garantieren, dies ist ein kundenorientierter Vorgang.

Bei einer Funktion, die als Container-Image definiert ist, wählen Sie beim Erstellen des Container-Images eine Laufzeit und die Linux-Distribution aus. Um die Laufzeit zu ändern, erstellen Sie ein neues Container-Image.

Wenn Sie ein ZIP-Dateiarchiv für das Bereitstellungspaket verwenden, wählen Sie beim Erstellen der Funktion eine Laufzeit aus. Um die Laufzeit zu ändern, können Sie die Konfiguration Ihrer Funktion aktualisieren. Die Laufzeit ist mit einer der Amazon Linux-Distributionen gepaart. Die zugrunde liegende Ausführungsumgebung bietet zusätzliche Bibliotheken und Umgebungsvariablen, auf die Sie über Ihren Funktionscode zugreifen können.

Lambda ruft Ihre Funktion in einer Ausführungsumgebung auf. Die Ausführungsumgebung bietet eine sichere und isolierte Laufzeitumgebung, die die zum Ausführen Ihrer Funktion erforderlichen Ressourcen verwaltet. Lambda verwendet die Ausführungsumgebung eines vorherigen Aufrufs wieder, sofern vorhanden, oder kann eine neue Ausführungsumgebung erstellen.

Um andere Sprachen in Lambda zu verwenden, wie Geh oder Rost, benutze ein OS-only Laufzeit. Die Lambda-Ausführungsumgebung bietet eine Laufzeitschnittstelle zum Abrufen von Aufrufereignissen und Senden von Antworten. Sie können andere Sprachen bereitstellen, indem Sie zusammen mit Ihrem Funktionscode oder in einem Layer eine benutzerdefinierte Laufzeit implementieren.

Unterstützte Laufzeiten

In der folgenden Tabelle sind die unterstützten Lambda-Laufzeiten für Event Functions und die voraussichtlichen Verfallsdaten aufgeführt. Wenn eine Laufzeit als veraltet gilt, können Sie für einen begrenzten Zeitraum weiterhin Funktionen erstellen und aktualisieren. Weitere Informationen finden Sie unter the section called “Unterstützung der Verwendung nach der Einstellung”. Die Tabelle enthält die aktuell angenommenen Ablaufdaten für die Laufzeit, basierend auf unserer the section called “Richtlinie für den Laufzeitablauf”. Diese Daten dienen zu Planungszwecken und können sich ändern.

> **Note**
>
> Die Node.js Die Laufzeiten 26 und Python 3.15 sind in öffentliche Vorschau. Vorschau-Laufzeiten fallen nicht unter das Lambda-SLA oder den technischen Support und sollten nicht für Produktionsworkloads verwendet werden.
> Der Zweck der Vorschau-Runtimes besteht darin, vor der allgemeinen Verfügbarkeit Feedback von Kunden, Partnern und Upstream-Sprachgemeinschaften einzuholen, während grundlegende Änderungen weiterhin möglich sind. Feedback können Sie über die dafür vorgesehene GitHub Probleme für Node.js 26 und Python 3.15. Weitere Informationen finden Sie hier Blogbeitrag.

> **Important**
>
> Das Ende der Lebensdauer von Amazon Linux 2 ist für den 30. Juni 2026 geplant. Lambda-Laufzeiten und Container-Basis-Images für Java 8 (AL2), Java 11 (AL2), Java 17 (AL2), Python 3.10, Python 3.11 und provided.al2 erhalten weiterhin Patches für kritisch und ausgewählt wichtig Sicherheitsprobleme bei Amazon Linux 2, zusätzlich zu Sprachlaufzeit-Patches, bis zu den in der folgenden Tabelle aufgeführten Verfallsdaten.
> Wir empfehlen Kunden, so bald wie möglich ein Upgrade auf eine auf Amazon Linux 2023 basierende Runtime durchzuführen. Für Kunden, die ein Upgrade auf Java 21 oder Java 25 durchführen, können Sie AWS Transform kundenspezifisch um bei diesen Upgrades zu helfen. Für Kunden, die ihre Java-Version nicht aktualisieren können, bieten wir Amazon Linux 2023-basierte Runtimes für Java 8, Java 11 und Java 17 an, die dem AL2023-Verfallszeitplan entsprechen.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 26	`nodejs26.x`	Amazon Linux 2023	Nicht geplant	Nicht geplant	Nicht geplant
Node.js 24	`nodejs24.x`	Amazon Linux 2023	30. April 2028	1. Juni 2028	1. Juli 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	30. April 2027	29. Juli 2027	31. August 2027
Python 3,15	`python3.15`	Amazon Linux 2023	Nicht geplant	Nicht geplant	Nicht geplant
Python 3.14	`python3.14`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.13	`python3.13`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 3.12	`python3.12`	Amazon Linux 2023	31. Oktober 2028	30. November 2028	10. Januar 2029
Python 3.11	`python3.11`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027
Python 3.10	`python3.10`	Amazon Linux 2	31. Oktober 2026	29. Juli 2027	31. August 2027
Java 25	`java25`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 21	`java21`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 17	`java17.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 11	`java11.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 8	`java8.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 17	`java17`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027
Java 11	`java11`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027
Java 8	`java8.al2`	Amazon Linux 2	30. Juni 2027	29. Juli 2027	31. August 2027

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 10	`dotnet10`	Amazon Linux 2023	14. November 2028	14. Dezember 2028	15. Januar 2029
.NET 9 (nur Container)	`dotnet9`	Amazon Linux 2023	10. November 2026	Nicht geplant	Nicht geplant
.NET 8	`dotnet8`	Amazon Linux 2023	10. November 2026	29. Juli 2027	31. August 2027
Rubin 4,0	`ruby4.0`	Amazon Linux 2023	31. März 2029	30. April 2029	31. Mai 2029
Rubin 3.4	`ruby3.4`	Amazon Linux 2023	31. März 2028	30. April 2028	31. Mai 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	31. März 2027	29. Juli 2027	31. August 2027
OS-only Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029

> **Note**
>
> Für neue Regionen wird Lambda keine Laufzeiten unterstützen, die innerhalb der nächsten sechs Monate veraltet sein werden.

Lambda hält verwaltete Laufzeiten und ihre entsprechenden Container-Basis-Images mit Patches und Unterstützung für kleinere Versionen auf dem neuesten Stand. Weitere Informationen finden Sie unter Lambda-Laufzeitaktualisierungen.

Um programmatisch mit anderen zu interagieren AWS-Services und Ressourcen aus Ihrer Funktion, können Sie eine von verwenden AWS SDKs. Die Node.js, Python- und Ruby-Laufzeiten enthalten eine Version von AWS SDK. Um jedoch die volle Kontrolle über Ihre Abhängigkeiten zu behalten und die Abwärtskompatibilität bei automatischen Laufzeitaktualisierungen zu maximieren, empfehlen wir Ihnen, die von Ihrem Code verwendeten SDK-Module (zusammen mit allen Abhängigkeiten) immer in das Bereitstellungspaket Ihrer Funktion oder in eine Lambda-Ebene aufzunehmen.

Wir empfehlen, die in der Laufzeit enthaltene SDK-Version nur zu verwenden, wenn Sie keine zusätzlichen Pakete in Ihre Bereitstellung aufnehmen können. Wenn Sie Ihre Funktion beispielsweise mit dem Code-Editor der Lambda-Konsole oder mithilfe von Inline-Funktionscode in einem erstellen CloudFormation Vorlage.

Lambda aktualisiert regelmäßig die Versionen von AWS SDKs, die in der enthalten sind Node.js, Python- und Ruby-Laufzeiten. Informationen darüber, welche Version des AWS -SDKs in der von Ihnen verwendeten Laufzeit enthalten ist, finden Sie in den folgenden Abschnitten:

- Runtime-included SDK-Versionen (Node.js)
- Runtime-included SDK-Versionen (Python)
- Runtime-included SDK-Versionen (Ruby)

Lambda unterstützt weiterhin die Programmiersprache Go, auch nachdem die Go 1.x-Laufzeit veraltet ist. Weitere Informationen finden Sie unter Migrieren AWS Lambda Funktionen aus dem Go1.x Laufzeit zur benutzerdefinierten Laufzeit auf Amazon Linux 2 auf der AWS Compute-Blog.

Alle unterstützten Lambda-Laufzeiten unterstützen sowohl x86_64- als auch arm64-Architekturen.

Neue Laufzeit-Versionen

Lambda stellt verwaltete Laufzeiten für neue Sprachversionen nur dann zur Verfügung, wenn die Version die LTS-Phase (Long-Term Support) des Veröffentlichungszyklus der Sprache erreicht. Zum Beispiel für die Node.js Release-Zyklus, wenn die Veröffentlichung die aktive LTS-Phase erreicht.

Bevor die Version die LTS-Phase erreicht, befindet sie sich noch in der Entwicklung und kann noch grundlegenden Änderungen unterliegen. Lambda wendet Laufzeit-Updates standardmäßig automatisch an. Wenn Sie also Änderungen an einer Laufzeitversion vornehmen, funktionieren Ihre Funktionen möglicherweise nicht mehr wie erwartet.

Lambda bietet keine verwalteten Laufzeiten für Sprachversionen, die nicht für die LTS-Veröffentlichung geplant sind.

Im Folgenden ist der geplante Startmonat für die kommenden Lambda-Laufzeiten aufgeführt. Diese Termine sind nur Richtwerte und können sich ändern.

- Node.js 26 - November 2026
- Python 3.15 - November 2026

Richtlinie für den Laufzeitablauf

Lambda-Laufzeiten für ZIP-Dateiarchive werden um eine Kombination aus Betriebssystem, Programmiersprache und Softwarebibliotheken aufgebaut, die Wartungen und Sicherheitsupdates erfordern. Die standardmäßige Verwertungsrichtlinie von Lambda sieht vor, eine Laufzeit als veraltet zu betrachten, wenn eine Hauptkomponente der Laufzeit das Ende des langfristigen Community-Supports (LTS) erreicht und keine Sicherheitsupdates mehr verfügbar sind. In den meisten Fällen handelt es sich dabei um die Sprachlaufzeit. In einigen Fällen kann eine Laufzeit jedoch als veraltet gelten, weil das Betriebssystem das LTS-Ende erreicht.

Nachdem eine Laufzeitumgebung veraltet ist, AWS wendet möglicherweise keine Sicherheitspatches oder Updates mehr auf diese Laufzeit an, und Funktionen, die diese Laufzeit verwenden, haben keinen Anspruch mehr auf technischen Support. Solche veralteten Runtimes werden „wie sie sind“ ohne jegliche Garantie zur Verfügung gestellt und können Bugs, Fehler, Defekte oder andere Sicherheitslücken enthalten.

Weitere Informationen zur Verwaltung von Runtime-Upgrades und veralteten Versionen finden Sie in den folgenden Abschnitten und Verwaltung AWS Lambda Laufzeit-Upgrades auf der AWS Compute-Blog.

> **Important**
>
> Lambda verzögert gelegentlich die Einstellung einer Lambda-Laufzeit für einen begrenzten Zeitraum über das Ende der Unterstützung der Sprachversion hinaus, die von der Laufzeit unterstützt wird. Während dieses Zeitraums wendet Lambda nur Sicherheits-Patches auf das Laufzeit-Betriebssystem an. Lambda wendet keine Sicherheitspatches auf Laufzeiten von Programmiersprachen an, nachdem das Support-Ende erreicht ist.

Modell der geteilten Verantwortung

Lambda ist für das Kuratieren und Veröffentlichen von Sicherheitsupdates für alle unterstützten verwalteten Laufzeiten und Container-Basisimages verantwortlich. Standardmäßig wendet Lambda diese Aktualisierungen automatisch auf Funktionen mit verwalteten Laufzeiten an. Wenn die Standardeinstellung für automatische Laufzeit-Updates geändert wurde, finden Sie weitere Informationen im Laufzeitmanagementkontrollen Modell der geteilten Verantwortung. Bei Funktionen, die mit Container-Images bereitgestellt werden, sind Sie dafür verantwortlich, das Container-Image Ihrer Funktion aus dem neuesten Basis-Image neu zu erstellen und das Container-Image erneut bereitzustellen.

Wenn eine Runtime veraltet ist, erlischt die Verantwortung von Lambda für die Aktualisierung der verwalteten Runtime und der Container-Basisimages. Sie sind dafür verantwortlich, Ihre Funktionen so zu aktualisieren, dass sie ein unterstütztes Laufzeit- oder Basis-Image verwenden.

In allen Fällen sind Sie dafür verantwortlich, Aktualisierungen an Ihrem Funktionscode, einschließlich seiner Abhängigkeiten, vorzunehmen. Ihre Verantwortlichkeiten im Rahmen des Modells der gemeinsamen Verantwortung sind in der folgenden Tabelle zusammengefasst.

Laufzeit-Lebenszyklusphase	Verantwortungen von Lambda	Ihre Aufgaben
Unterstützte verwaltete Laufzeit	Stellen Sie regelmäßige Laufzeit-Updates mit Sicherheitspatches und anderen Updates bereit. Wenden Sie Laufzeit-Updates standardmäßig automatisch an (Informationen zu nicht standardmäßigen Verhaltensweisen finden Sie unter the section called “Modi der Laufzeitaktualisierung”).	Aktualisieren Sie Ihren Funktionscode, einschließlich der Abhängigkeiten, um Schwachstellen zu schließen.
Unterstütztes Container-Image	Regelmäßige Aktualisierung des Container-Basis-Images mit Sicherheitspatches und anderen Updates.	Aktualisieren Sie Ihren Funktionscode, einschließlich der Abhängigkeiten, um Schwachstellen zu schließen.

Laufzeit-Lebenszyklusphase	Verantwortungen von Lambda	Ihre Aufgaben
		Erstellen Sie Ihr Container-Image regelmäßig neu und stellen Sie es erneut bereit, indem Sie das neueste Basis-Image verwenden.
Verwaltete Laufzeit nähert sich dem Verfall	Informieren Sie die Kunden anhand der Dokumenta tion, bevor die Laufzeit nicht mehr unterstützt wird. Health Dashboard, E-Mail und Trusted Advisor. Die Verantwortung für Laufzeit-Updates endet mit dem Zeitpunkt, an dem sie veraltet sind.	Überwachen Sie die Lambda-Dokumentation, Health Dashboard, E-Mail oder Trusted Advisor für Informati onen zur abgelaufenen Laufzeit. Führen Sie ein Upgrade von Funktionen auf eine unterstüt zte Laufzeit durch, bevor die vorherige Laufzeit veraltet ist.
Container-Image nähert sich der Veralterung	Benachrichtigungen über veraltete Versionen sind für Funktionen, die Container -Images verwenden, nicht verfügbar. Die Verantwortung für Container-Basis-Images-Upda tes endet mit dem Zeitpunkt, an dem sie veraltet sind.	Beachten Sie die Zeitpläne für veraltete Versionen und aktualisieren Sie Funktionen auf ein unterstütztes Basis-Image, bevor das vorherige Image veraltet ist.

Unterstützung der Verwendung nach der Einstellung

Nachdem eine Laufzeit veraltet ist, AWS wendet möglicherweise keine Sicherheitspatches oder Updates mehr auf diese Laufzeit an, und Funktionen, die diese Laufzeit verwenden, haben keinen Anspruch mehr auf technischen Support. Sie können Ihre Funktionen zwar auf unbestimmte Zeit weiterhin aufrufen, AWS empfiehlt dringend, auf eine unterstützte Laufzeit zu migrieren.

Veraltete Laufzeiten werden „wie sie sind“ ohne jegliche Garantie bereitgestellt und können Bugs, Fehler, Defekte oder andere Schwachstellen enthalten. Bei Funktionen, die eine veraltete Laufzeit verwenden, kann es zu Leistungseinbußen oder anderen Problemen kommen, wie z. B. dem Ablauf eines Zertifikats, was dazu führen kann, dass sie nicht mehr richtig funktionieren.

Sie können eine Funktion so aktualisieren, dass sie jederzeit eine neuere unterstützte Laufzeit verwendet, nachdem eine Laufzeit veraltet ist. Wir empfehlen, Ihre Funktion mit der neuen Laufzeit zu testen, bevor Sie Änderungen in Produktionsumgebungen vornehmen. Sie können nicht zur veralteten Laufzeit zurückkehren, nachdem Funktionsaktualisierungen blockiert wurden. Wir empfehlen die Verwendung von Funktionsversionen und Aliasnamen, um eine sichere Bereitstellung mit Rollback zu ermöglichen.

Die folgende Zeitleiste beschreibt, was geschieht, wenn eine Laufzeit veraltet ist:

Laufzeit-Lebenszyklusphase	Wann	Was
Frist für Hinweis zur Einstellung	Mindestens 180 Tage vor Ablauf der Einstellung	• AWS sendet Benachrichtigungen per E-Mail und der Health Dashboard an Konten, die Funktionen haben, die diese Laufzeit verwenden `$LATEST` Version. • Betroffene Funktionen sind auch auf der Registerkarte mit Health Dashboard – Geplante Änderungen und in der Prüfung für AWS Trusted Advisor – Veraltete Laufzeiten aufgeführt.
Ablehnung	Datum der Veraltung	• AWS wendet möglicherweise keine Sicherheitsupdates oder andere Updates mehr an. • Funktionen sind nicht mehr für den technischen Support berechtigt. • Sie können mit der veralteten Laufzeit in der Lambda-Konsole keine Funktionen mehr erstellen oder aktualisieren. Sie können weiterhin Funktionen über das erstellen und aktualisieren AWS CLI, AWS SAM, oder CloudFormation.

Laufzeit-Lebenszyklusphase	Wann	Was
Blockfunktion erstellen	Mindestens 30 Tage nach der Einstellung	• Lambda beginnt, die Erstellung neuer Funktionen zu blockieren. • Sie können den Code und die Konfiguration für bestehende Funktionen weiterhin über das aktualisieren AWS CLI, AWS SAM, oder CloudFormation..
Blockfunktion aktualisieren	Mindestens 60 Tage nach der Einstellung	• Lambda beginnt, die Aktualisierung des Codes und der Konfiguration für bestehende Funktionen zu blockieren. • Sie können die Konfiguration der Funktion immer noch auf eine unterstützte Laufzeit aktualisieren. Ein Rollback zur veralteten Laufzeit kann jedoch blockiert werden.

> **Note**
>
> Für einige Laufzeiten AWS verschiebt die Termine block-function-create und block-function-update über die üblichen 30 und 60 Tage nach der Deprecation hinaus. AWS hat diese Änderung als Reaktion auf Kundenfeedback vorgenommen, um Ihnen mehr Zeit für die Aktualisierung Ihrer Funktionen zu geben. In den Tabellen in the section called “Unterstützte Laufzeiten” und the section called “Veraltete Laufzeitumgebungen” finden Sie die Daten für Ihre Laufzeit. Lambda beginnt nicht vor den in diesen Tabellen angegebenen Terminen mit dem Blockieren von Funktionserstellungen oder -aktualisierungen.

Empfangen von Benachrichtigungen über veraltete Laufzeitversionen

Wenn sich eine Laufzeitumgebung ihrem Verfallsdatum nähert, sendet Lambda Ihnen eine E-Mail-Benachrichtigung, falls Funktionen in Ihrem AWS-Konto verwende diese Laufzeit. Benachrichtigungen werden auch im angezeigt Health Dashboard und in AWS Trusted Advisor.

- Empfangen von E-Mail-Benachrichtigungen:

Lambda sendet Ihnen mindestens 180 Tage, bevor eine Laufzeit veraltet ist, eine E-Mail-Warnung. In dieser E-Mail sind die $LATEST-Versionen aller Funktionen aufgeführt, die die Laufzeit verwenden. Um eine vollständige Liste der betroffenen Funktionsversionen anzuzeigen, verwenden Sie Trusted Advisor oder siehe the section called “Daten über Funktionen zur Laufzeit abrufen”.

Lambda sendet eine E-Mail-Benachrichtigung an Ihre AWS-Konto der primäre Kontokontakt. Informationen zum Anzeigen oder Aktualisieren der E-Mail-Adressen in Ihrem Konto finden Sie in der allgemeinen AWS -Referenz unter Updating contact information.

- Empfangen von Benachrichtigungen über den Health Dashboard:

  Das Health Dashboard zeigt mindestens eine Benachrichtigung an 180 Tage bevor eine Runtime veraltet ist. Benachrichtigungen werden auf der Seite Ihr Kontostatus unter Andere Benachrichtigungen angezeigt. Auf der Registerkarte Betroffene Ressourcen in der Benachrichtigung sind die $LATEST-Versionen aller Funktionen aufgeführt, die die Laufzeit verwenden.

  > **Note**
  >
  > Eine vollständige und aktuelle Liste der betroffenen Funktionsversionen finden Sie unter Trusted Advisor oder siehe the section called “Daten über Funktionen zur Laufzeit abrufen”.

  Health Dashboard Benachrichtigungen laufen 90 Tage ab, nachdem die betroffene Laufzeit veraltet ist.

- Verwenden AWS Trusted Advisor

  Trusted Advisor zeigt mindestens eine Benachrichtigung an 180 Tage bevor eine Runtime veraltet ist. Benachrichtigungen werden auf der Seite Sicherheit angezeigt. Eine Liste der betroffenen Funktionen wird unter AWS Lambda -Funktionen, die veraltete Laufzeiten verwenden angezeigt. Diese Liste von Funktionen zeigt sowohl die $LATEST als auch veröffentlichte Versionen und wird automatisch aktualisiert, um den aktuellen Status Ihrer Funktionen widerzuspiegeln.

  Sie können wöchentliche E-Mail-Benachrichtigungen von aktivieren Trusted Advisor in der Einstellungen Seite der Trusted Advisor Konsole.

Veraltete Laufzeitumgebungen

Die folgenden Laufzeiten haben das Ende der Unterstützung erreicht:

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
OS-only Laufzeit	`provided.al2`	Amazon Linux 2	31. Juli 2026	29. Juli 2027	31. August 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	30. April 2026	29. Juli 2027	31. August 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	31. März 2026	29. Juli 2027	31. August 2027
Python 3.9	`python3.9`	Amazon Linux 2	15. Dezember 2025	29. Juli 2027	31. August 2027
Node.js 18	`nodejs18.x`	Amazon Linux 2	1. Sept. 2025	29. Juli 2027	31. August 2027
.NET 6	`dotnet6`	Amazon Linux 2	20. Dezember 2	29. Juli 2027	31. August 2027
Python 3.8	`python3.8`	Amazon Linux 2	14. Oktober 202	29. Juli 2027	31. August 2027
Node.js 16	`nodejs16.x`	Amazon Linux 2	12. Juni 2024	29. Juli 2027	31. August 2027
.NET 7 (nur Container)	`dotnet7`	Amazon Linux 2	14. Mai 2024	N/A	N/A
Java 8	`java8`	Amazon Linux	8. Januar 2024	8. Februar 2024	31. August 2027

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Go 1.x	`go1.x`	Amazon Linux	8. Januar 2024	8. Februar 2024	31. August 2027
OS-only Laufzeit	`provided`	Amazon Linux	8. Januar 2024	8. Februar 2024	31. August 2027
Ruby 2.7	`ruby2.7`	Amazon Linux 2	7. Dezember 20	9. Januar 2024	31. August 2027
Node.js 14	`nodejs14.x`	Amazon Linux 2	4. Dezember 20	9. Januar 2024	31. August 2027
Python 3.7	`python3.7`	Amazon Linux	4. Dezember 20	9. Januar 2024	31. August 2027
.NET Core 3.1	`dotnetcore3.1`	Amazon Linux 2	3. Apr 2023	3. Apr 2023	3. Mai 2023
Node.js 12	`nodejs12.x`	Amazon Linux 2	31. März 2023	31. März 2023	30. Apr 2023
Python 3.6	`python3.6`	Amazon Linux	18. Juli 2022	18. Juli 2022	29. August 2022
.NET 5 (nur Container)	`dotnet5.0`	Amazon Linux 2	10. Mai 2022	N/A	N/A
.NET Core 2.1	`dotnetcore2.1`	Amazon Linux	5. Januar 2022	5. Januar 2022	13. Apr 2022
Node.js 10	`nodejs10.x`	Amazon Linux 2	30. Juli 2021	30. Juli 2021	14. Februar 2022
Ruby 2.5	`ruby2.5`	Amazon Linux	30. Juli 2021	30. Juli 2021	31. März 2022

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 2.7	`python2.7`	Amazon Linux	15. Juli 2021	15. Juli 2021	30. Mai 2022
Node.js 8,10	`nodejs8.10`	Amazon Linux	6. März 2020	4. Februar 2020	6. März 2020
Node.js 4.3	`nodejs4.3`	Amazon Linux	5. März 2020	3. Februar 2020	5. März 2020
Node.js 4.3 Rand	`nodejs4.3-edge`	Amazon Linux	5. März 2020	31. März 2019	30. Apr 2019
Node.js 6,10	`nodejs6.10`	Amazon Linux	12. August 201	12. Juli 2019	12. August 2019
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	27. Juni 2019	30. Juni 2019	30. Juli 2019
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	30. Mai 2019	30. Apr 2019	30. Mai 2019
Node.js 0,10	`nodejs`	Amazon Linux	30. August 2016	30. September 2016	31. Oktober 2016

In den meisten Fällen ist das Ende der Nutzungsdauer einer Sprachversion oder eines Betriebssystems weit im Voraus bekannt. Die folgenden Links enthalten Zeitpläne für das Ende der Lebensdauer für jede Sprache, die Lambda als verwaltete Laufzeit unterstützt.

Richtlinien für die Unterstützung von Sprache und Framework-Bedingungen

- Node.js — github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java – www.oracle.com and Corretto FAQs

- Go – golang.org
- .NET – dotnet.microsoft.com

Verstehen, wie Lambda Laufzeit-Versionsupdates verwaltet

Lambda hält jede verwaltete Laufzeit mit Sicherheitsupdates, Fehlerbehebungen, neuen Funktionen, Leistungsverbesserungen und Unterstützung für Nebenversionen auf dem neuesten Stand. Diese Laufzeitaktualisierungen werden als Laufzeitversionen veröffentlicht. Lambda wendet Laufzeitaktualisierungen auf Event Functions an, indem es die Funktion von einer früheren Runtime-Version auf eine neue Runtime-Version migriert.

Für Funktionen, die verwaltete Laufzeiten verwenden, wendet Lambda Laufzeitaktualisierungen standardmäßig automatisch an. Mit automatischen Laufzeitaktualisierungen übernimmt Lambda den operativen Aufwand für das Patchen der Laufzeitversionen. Für die meisten Kunden sind automatische Aktualisierungen die richtige Wahl. Sie können dieses Standardverhalten ändern, indem Sie die Einstellungen für die Laufzeitverwaltung konfigurieren.

Lambda veröffentlicht auch jede neue Laufzeitversion als Container-Image. Um Laufzeitversionen für Container-basierte Funktionen zu aktualisieren, müssen Sie ein neues Container-Image aus dem aktualisierten Basis-Image erstellen und Ihre Funktion erneut bereitstellen.

Jeder Laufzeitversion ist eine Versionsnummer und ein ARN (Amazon Resource Name) zugeordnet. Laufzeitversionsnummern verwenden ein von Lambda definiertes Nummerierungsschema, unabhängig von den Versionsnummern, die die Programmiersprache verwendet. Laufzeit-Versionsnummern sind nicht immer fortlaufend. Auf Version 42 könnte beispielsweise Version 45 folgen. Der Laufzeitversions-ARN ist eine eindeutige Kennung für jede Laufzeitversion. Sie können den ARN der aktuellen Laufzeitversion Ihrer Funktion in der Lambda-Konsole oder in der `INIT_START`-Zeile der Funktionsprotokolle anzeigen. Hinweise zur Verwaltung von Runtime-Updates in gesundheitskritischen Anwendungen finden Sie unter. the section called “Berechtigungen”

Laufzeitversionen sollten nicht mit Laufzeitkennungen verwechselt werden. Jede Laufzeit hat eine eindeutige Laufzeitkennung, z. B. `python3.14` oder `nodejs24.x`. Diese entsprechen den jeweiligen Hauptversionen der Programmiersprachen. Laufzeitversionen beschreiben die Patch-Version einer einzelnen Laufzeit.

> **Note**
>
> Der ARN für dieselbe Runtime-Versionsnummer kann je nach AWS-Regionen CPU-Architektur variieren.

Themen

- Abwärtskompatibilität
- Modi der Laufzeitaktualisierung
- Two-phase Einführung der Laufzeitversion
- Konfigurieren der Einstellungen für die Lambda-Laufzeitverwaltung
- Rollback einer Lambda-Laufzeitversion
- Identifizieren von Änderungen der Lambda-Laufzeitversion
- Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit
- Kontrolle der Lambda-Laufzeitaktualisierungsberechtigungen für Anwendungen mit hoher Konformität

Abwärtskompatibilität

Lambda ist bestrebt, Laufzeitaktualisierungen bereitzustellen, die mit vorhandenen Funktionen abwärtskompatibel sind. Wie beim Software-Patching gibt es jedoch seltene Fälle, in denen sich eine Laufzeitaktualisierung negativ auf eine vorhandene Funktion auswirken kann. Beispielsweise können Sicherheits-Patches ein zugrunde liegendes Problem mit einer vorhandenen Funktion aufdecken, das vom vorherigen, unsicheren Verhalten abhängt.

Beim Erstellen und Bereitstellen Ihrer Funktion ist es wichtig zu verstehen, wie Sie Ihre Abhängigkeiten verwalten, um mögliche Inkompatibilitäten mit einem zukünftigen Laufzeit-Update zu vermeiden. Nehmen wir zum Beispiel an, dass Ihre Funktion von Paket A abhängig ist, das wiederum von Paket B abhängt. Beide Pakete sind in der Lambda-Laufzeit enthalten (sie könnten beispielsweise Teile des SDK oder seiner Abhängigkeiten oder Teile der Laufzeitsystembibliotheken sein).

Betrachten Sie folgende Szenarien:

Bereitstellung	Patching-kompatibel	Grund
• Paket A: Ab Laufzeit verwenden • Paket B: Ab Laufzeit verwenden	Ja	Zukünftige Laufzeit-Updates für die Pakete A und B sind abwärtskompatibel.
• Paket A: Im Bereitste llungspaket	Ja	Ihre Bereitstellung hat Vorrang, sodass zukünftig

Bereitstellung	Patching-kompatibel	Grund
• Paket B: Im Bereitstellungspaket		e Laufzeit-Updates für die Pakete A und B keine Auswirkungen haben.
• Paket A: Im Bereitstellungspaket • Paket B: Ab Laufzeit verwenden	Ja*	Zukünftige Laufzeit-Updates für das Paket B sind abwärtskompatibel. * Wenn A und B eng miteinander verbunden sind, können Kompatibilitätsprobleme auftreten. Beispielsweise sollten die `botocore` Pakete `boto3` und im AWS SDK für Python zusammen bereitgestellt werden.
• Paket A: Ab Laufzeit verwenden • Paket B: Im Bereitstellungspaket	Nein	Zukünftige Laufzeit-Updates für Paket A erfordern möglicherweise eine aktualisierte Version von Paket B. Die bereitgestellte Version von Paket B hat jedoch Vorrang und ist möglicherweise nicht mit der aktualisierten Version von Paket A aufwärtskompatibel.

Befolgen Sie die folgenden bewährten Methoden, um die Kompatibilität mit zukünftigen Laufzeit-Updates aufrechtzuerhalten:

- Wenn möglich, packen Sie alle Abhängigkeiten: Nehmen Sie alle erforderlichen Bibliotheken, einschließlich des AWS SDK und seiner Abhängigkeiten, in Ihr Bereitstellungspaket auf. Dadurch wird ein stabiler, kompatibler Satz von Komponenten gewährleistet.

- Verwenden Sie zur Laufzeit bereitgestellte SDKs sparsam: Verlassen Sie sich nur dann auf das von der Laufzeit bereitgestellte SDK, wenn Sie keine zusätzlichen Pakete hinzufügen können (z. B. wenn Sie den Lambda-Konsolen-Code-Editor oder Inline-Code in einer Vorlage verwenden). AWS CloudFormation
- Vermeiden Sie das Überschreiben von Systembibliotheken: Stellen Sie keine benutzerdefinierten Betriebssystembibliotheken bereit, die mit future Runtime-Updates in Konflikt geraten könnten.

Modi der Laufzeitaktualisierung

Lambda ist bestrebt, Laufzeitaktualisierungen bereitzustellen, die mit vorhandenen Funktionen abwärtskompatibel sind. Wie beim Software-Patching gibt es jedoch seltene Fälle, in denen sich eine Laufzeitaktualisierung negativ auf eine vorhandene Funktion auswirken kann. Beispielsweise können Sicherheits-Patches ein zugrunde liegendes Problem mit einer vorhandenen Funktion aufdecken, das vom vorherigen, unsicheren Verhalten abhängt. Lambda-Laufzeitverwaltungskontrollen tragen dazu bei, das Risiko von Auswirkungen auf Ihre Workloads im seltenen Fall einer Inkompatibilität einer Laufzeitversion zu verringern. Für jede Funktionsversion ($LATEST oder veröffentlichte Version) können Sie einen der folgenden Laufzeitaktualisierungsmodi wählen:

- Automatisch (Standard) – Automatische Aktualisierung auf die neueste und sicherste Laufzeitversion mit Two-phase Einführung der Laufzeitversion. Diesen Modus empfehlen wir den meisten Kunden, damit Sie immer von Laufzeitaktualisierungen profitieren.
- Funktion aktualisieren – Aktualisieren Sie die Laufzeit Ihrer Funktion auf die neueste und sicherste Laufzeitversion. Wenn Sie Ihre Funktion aktualisieren, aktualisiert Lambda die Laufzeit Ihrer Funktion auf die neueste und sicherste Laufzeitversion. Dieser Ansatz synchronisiert Laufzeitaktualisierungen mit Funktionsbereitstellungen, sodass Sie die Kontrolle darüber haben, wann Lambda Laufzeitaktualisierungen anwendet. In diesem Modus können Sie seltene Inkompatibilitäten bei Laufzeitaktualisierungen frühzeitig erkennen und beheben. Wenn Sie diesen Modus verwenden, müssen Sie Ihre Funktionen regelmäßig aktualisieren, um deren Laufzeit auf dem neuesten Stand zu halten.
- Manuell – Aktualisieren Sie Ihre Laufzeitversion manuell. Sie geben in Ihrer Funktionskonfiguration eine Laufzeitversion an. Die Funktion verwendet diese Laufzeitversion unbegrenzt.

  In dem seltenen Fall, dass eine neue Laufzeitversion mit einer vorhandenen Funktion nicht kompatibel ist, können Sie diesen Modus verwenden, um Ihre Funktion auf eine frühere Laufzeitversion zurückzusetzen. Wir raten davon ab, den Modus Manual (Manuell) zu verwenden,

um Laufzeitkonsistenz über Bereitstellungen hinweg zu erreichen. Weitere Informationen finden Sie unter Rollback einer Lambda-Laufzeitversion.

Die Verantwortung für das Anwenden von Laufzeitaktualisierungen auf Ihre Funktionen hängt davon ab, welchen Laufzeitaktualisierungsmodus Sie auswählen. Weitere Informationen finden Sie unter Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit.

Two-phase Einführung der Laufzeitversion

Lambda führt neue Laufzeitversionen in der folgenden Reihenfolge ein:

1. In der ersten Phase wendet Lambda die neue Laufzeitversion an, sobald Sie eine Funktion erstellen oder aktualisieren. Eine Funktion wird aktualisiert, wenn Sie die UpdateFunctionConfigurationAPI-Operationen UpdateFunctionCodeoder aufrufen.
2. In der zweiten Phase aktualisiert Lambda alle Funktionen, die den Auto (Automatischen) Laufzeitaktualisierungsmodus verwenden und die noch nicht auf die neue Laufzeitversion aktualisiert wurden.

Die Gesamtdauer des Rollout-Prozesses hängt von mehreren Faktoren ab, einschließlich des Schweregrads von Sicherheits-Patches, die in der Laufzeitaktualisierung enthalten sind.

Wenn Sie Ihre Funktionen aktiv entwickeln und bereitstellen, erhalten Sie höchstwahrscheinlich in der ersten Phase neue Runtime-Versionen. Dadurch werden Laufzeitaktualisierungen mit Funktionsaktualisierungen synchronisiert. In dem seltenen Fall, dass sich die neueste Runtime-Version negativ auf Ihre Anwendung auswirkt, bietet Ihnen dieser Ansatz die Möglichkeit, umgehend Korrekturmaßnahmen zu ergreifen. Funktionen, die sich nicht in der aktiven Entwicklung befinden, erhalten auch in der zweiten Phase den operativen Nutzen von automatischen Laufzeitaktualisierungen.

Dieser Ansatz wirkt sich nicht auf Funktionen aus, die auf Function update (Funktionsaktualisierung) oder Manual (Manuell) festgelegt sind. Funktionen, die den Modus Function update (Funktionsaktualisierung) verwenden, erhalten die neuesten Laufzeitaktualisierungen nur, wenn Sie sie erstellen oder aktualisieren. Funktionen, die den Modus Manual (Manuell) verwenden, erhalten keine Laufzeitaktualisierungen.

Lambda veröffentlicht neue Laufzeitversionen schrittweise und fortlaufend in allen AWS-Regionen. Wenn Ihre Funktionen auf den Modus Automatische Aktualisierung oder Funktionsupdate eingestellt sind, ist es möglich, dass Funktionen, die zur gleichen Zeit in verschiedenen Regionen oder zu

unterschiedlichen Zeiten in derselben Region bereitgestellt werden, unterschiedliche Runtime-Versionen übernehmen. Kunden, die eine garantierte Konsistenz der Laufzeitversionen in ihren Umgebungen benötigen, sollten Container-Images verwenden, um ihre Funktionen bereitzustellen. Der Modus Manuell ist als vorübergehende Gegenmaßnahme gedacht, um im seltenen Fall eines Laufzeitzeitproblems ein Laufzeit-Rollback zu ermöglichen, dass eine Laufzeitversion nicht mit Ihrer Funktion kompatibel ist.

Konfigurieren der Einstellungen für die Lambda-Laufzeitverwaltung

Sie können die Laufzeitverwaltungseinstellungen für Ihre Lambda-Ereignisfunktionen mithilfe der Lambda-Konsole oder der AWS Command Line Interface ()AWS CLI konfigurieren.

> **Note**
>
> Sie können die Laufzeitverwaltungseinstellungen für jede Funktionsversion separat konfigurieren.

So konfigurieren Sie, wie Lambda Ihre Laufzeitversion aktualisiert (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Event-Funktion.
3. Wählen Sie auf der Registerkarte Code unter Runtime settings (Laufzeiteinstellungen) die Option Edit runtime management configuration (Laufzeitverwaltungskonfiguration bearbeiten) aus.
4. Wählen Sie unter Runtime management configuration (Laufzeitverwaltungskonfiguration) eine der folgenden Optionen aus:

   - Um Ihre Funktion automatisch auf die neueste Laufzeitversion aktualisieren zu lassen, wählen Sie Auto (Automatisch).
   - Um Ihre Funktion auf die neueste Laufzeitversion zu aktualisieren, wenn Sie die Funktion ändern, wählen Sie Function update (Funktionsaktualisierung).
   - Damit Ihre Funktion nur dann auf die neueste Laufzeitversion aktualisiert wird, wenn Sie den Laufzeitversions-ARN ändern, wählen Sie Manual (Manuell). Sie finden den Laufzeitversions-ARN unter Runtime management configuration (Laufzeitverwaltungskonfiguration). Sie finden den ARN auch in der `INIT_START`-Zeile Ihrer Funktionsprotokolle.

   Weitere Informationen zu diesen Optionen finden Sie unter Laufzeitaktualisierungsmodi.

5. Wählen Sie Speichern.

So konfigurieren Sie, wie Lambda Ihre Laufzeitversion aktualisiert (AWS CLI)

Um die Laufzeitverwaltung für eine Funktion zu konfigurieren, führen Sie den Befehl put-runtime-management-config AWS CLI aus. Wenn Sie den `Manual`-Modus verwenden, müssen Sie auch den ARN der Laufzeitversion angeben.

```
aws lambda put-runtime-management-config \
  --function-name my-function \
  --update-runtime-on Manual \
  --runtime-version-arn arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
  "RuntimeVersionArn": "arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

Rollback einer Lambda-Laufzeitversion

In dem seltenen Fall, dass eine neue Runtime-Version nicht mit Ihrer vorhandenen Event-Funktion kompatibel ist, können Sie ihre Runtime-Version auf eine frühere Version zurücksetzen. Dadurch bleibt Ihre Anwendung funktionsfähig und die Unterbrechung wird minimiert. Gleichzeitig wird Zeit bereitgestellt, um die Inkompatibilität zu beheben, bevor Sie zur neuesten Laufzeitversion zurückkehren.

Lambda legt keine zeitliche Begrenzung fest, wie lange Sie eine bestimmte Laufzeitversion verwenden können. Wir empfehlen jedoch dringend, so schnell wie möglich auf die neueste Laufzeitversion zu aktualisieren, um von den neuesten Sicherheits-Patches, Leistungsverbesserungen und Funktionen zu profitieren. Lambda bietet die Option, auf eine frühere Laufzeitversion zurückzusetzen, nur als vorübergehende Abhilfe für den seltenen Fall eines Kompatibilitätsproblems bei Laufzeitaktualisierungen. Bei Funktionen, die über einen längeren Zeitraum eine frühere Runtime-Version verwenden, kann es irgendwann zu Leistungseinbußen oder

Problemen kommen, z. B. wenn das Zertifikat abläuft, was dazu führen kann, dass sie nicht mehr richtig funktionieren.

Sie können eine Laufzeitversion auf die folgenden Arten zurücksetzen:

- Verwenden des Manual (Manuellen) Laufzeitaktualisierungsmodus
- Verwenden veröffentlichter Funktionsversionen

Weitere Informationen finden Sie im AWS Compute-Blog unter Einführung von AWS Lambda Runtime-Management-Steuerelementen.

Rollback einer Laufzeitversion mit dem Manual (Manuellen) Laufzeitaktualisierungsmodus

Wenn Sie den Auto (Automatischen) Aktualisierungsmodus für die Laufzeitversion oder die `$LATEST`-Laufzeitversion verwenden, können Sie Ihre Laufzeitversion im Manual (Manuellen) Modus zurücksetzen. Ändern Sie für die Funktionsversion, die Sie zurücksetzen möchten, den Aktualisierungsmodus der Laufzeitversion zu Manual (Manuell) und geben Sie den ARN der vorherigen Laufzeitversion an. Weitere Informationen zum Ermitteln des ARN der vorherigen Laufzeitversion finden Sie unter Identifizieren von Änderungen der Lambda-Laufzeitversion.

> **Note**
>
> Wenn die `$LATEST`-Version Ihrer Funktion für den Modus Manual (Manuell) konfiguriert ist, können Sie die von Ihrer Funktion verwendete CPU-Architektur oder Laufzeitversion nicht ändern. Um diese Änderungen vorzunehmen, müssen Sie in den Modus Auto (Automatisch) oder Function update (Funktionsaktualisierung) wechseln.

Rollback einer Laufzeitversion mit veröffentlichten Funktionsversionen

Veröffentlichte Funktionsversionen sind eine unveränderliche Momentaufnahme des `$LATEST`-Funktionscodes und der Konfiguration zum Zeitpunkt ihrer Erstellung. Im Modus Auto (Automatisch) aktualisiert Lambda während der zweiten Phase des Rollouts der Laufzeitversion automatisch die Laufzeitversion der veröffentlichten Funktionsversionen. Im Modus Function update (Funktionsaktualisierung) aktualisiert Lambda die Laufzeitversion veröffentlichter Funktionsversionen nicht.

Veröffentlichte Funktionsversionen, die den Modus Function update (Funktionsaktualisierung) verwenden, erstellen daher einen statischen Snapshot des Funktionscodes, der Konfiguration

und der Laufzeitversion. Wenn Sie den Modus Function update (Funktionsaktualisierung) mit Funktionsversionen verwenden, können Sie Laufzeitaktualisierungen mit Ihren Bereitstellungen synchronisieren. Sie können auch das Rollback von Code-, Konfigurations- und Laufzeitversionen koordinieren, indem Sie den Datenverkehr auf eine zuvor veröffentlichte Funktionsversion umleiten.

Sie können diesen Ansatz in Ihre Programme Continuous Integration und Continuous Delivery (CI/CD) integrieren, um ein vollautomatisches Rollback für den seltenen Fall einer Inkompatibilität von Runtime-Updates zu gewährleisten. Wenn Sie diesen Ansatz verwenden, müssen Sie Ihre Funktion regelmäßig aktualisieren und neue Funktionsversionen veröffentlichen, um die neuesten Laufzeitaktualisierungen zu erhalten. Weitere Informationen finden Sie unter Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit.

Identifizieren von Änderungen der Lambda-Laufzeitversion

Die Laufzeitversionsnummer und der ARN werden in der `INIT_START` Protokollzeile protokolliert, die Lambda jedes Mal, wenn es eine neue Ausführungsumgebung erstellt, an CloudWatch Logs ausgibt. Da die Ausführungsumgebung für alle Funktionsaufrufe dieselbeLaufzeit verwendet, gibt Lambda die `INIT_START`-Protokollzeile nur aus, wenn Lambda die Init-Phase ausführt. Lambda gibt diese Protokollzeile nicht für jeden Funktionsaufruf aus. Lambda gibt die Protokollzeile an CloudWatch Logs aus, sie ist jedoch in der Konsole nicht sichtbar.

> **Note**
>
> Laufzeit-Versionsnummern sind nicht immer fortlaufend. Auf Version 42 könnte beispielsweise Version 45 folgen.

Example Beispiel für die INIT_START-Protokollzeile

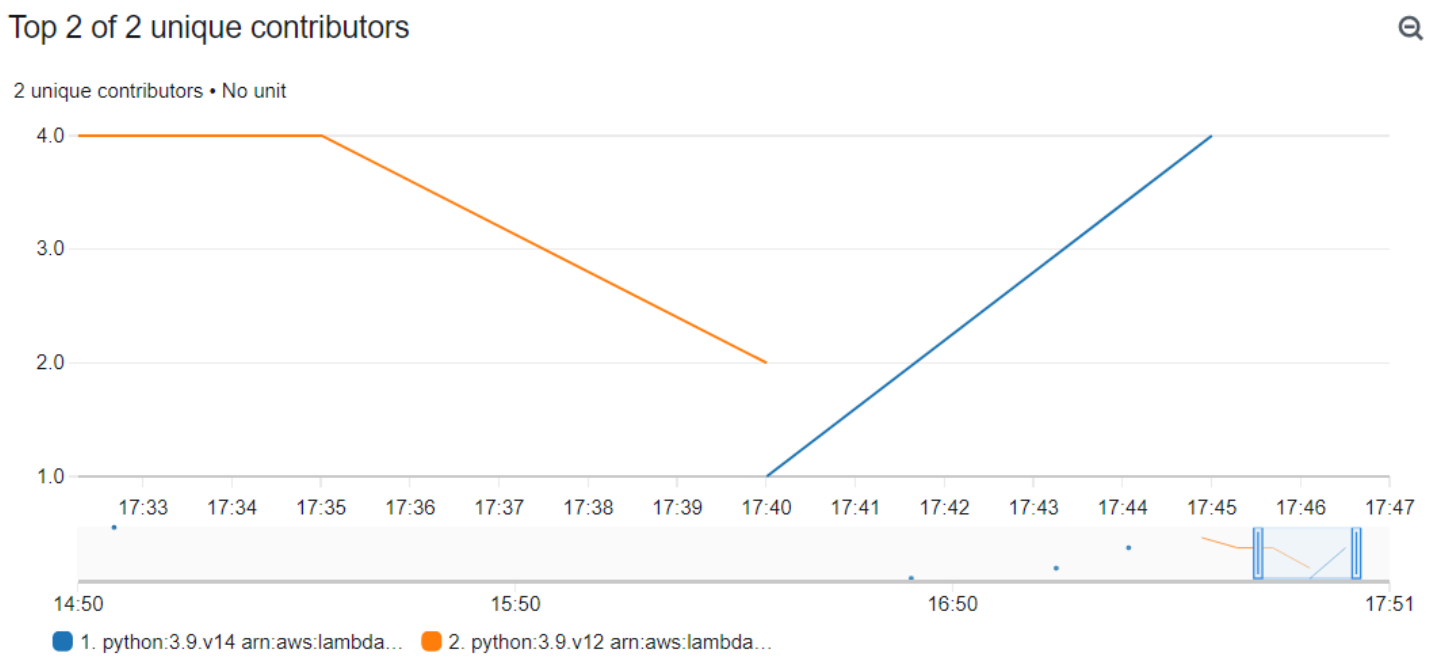
```
INIT_START Runtime Version: python:3.13.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

Anstatt direkt mit den Protokollen zu arbeiten, können Sie Amazon CloudWatch Contributor Insights verwenden, um Übergänge zwischen Runtime-Versionen zu identifizieren. Die folgende Regel zählt die verschiedenen Laufzeitversionen aus jeder `INIT_START`-Protokollzeile. Um die Regel zu verwenden, ersetzen Sie den Beispiel-Protokollgruppennamen `/aws/lambda/*` durch das entsprechende Präfix für Ihre Funktion oder Funktionsgruppe.

```
{
```

```
  "Schema": {
    "Name": "CloudWatchLogRule",
    "Version": 1
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

Der folgende CloudWatch Contributor Insights-Bericht zeigt ein Beispiel für einen Übergang zu einer Runtime-Version, wie er in der vorherigen Regel erfasst wurde. Die orange Linie zeigt die Initialisierung der Ausführungsumgebung für die frühere Laufzeitversion (python:3.13.v12), und die blaue Linie zeigt die Initialisierung der Ausführungsumgebung für die neue Laufzeitversion (python:3.13.v14).



Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit

Lambda ist für das Kuratieren und Veröffentlichen von Sicherheitsupdates für alle unterstützten verwalteten Laufzeiten und Container-Images verantwortlich. Die Verantwortung für die Aktualisierung vorhandener Eventfunktionen zur Verwendung der neuesten Runtime-Version hängt davon ab, welchen Runtime-Aktualisierungsmodus Sie verwenden.

Lambda ist dafür verantwortlich, Laufzeitaktualisierungen auf alle Funktionen anzuwenden, die für den Laufzeitaktualisierungsmodus Auto (Automatisch) konfiguriert sind.

Für Funktionen, die mit dem Laufzeitaktualisierungsmodus Function update (Funktionsaktualisierung) konfiguriert sind, sind Sie für die regelmäßige Aktualisierung Ihrer Funktion verantwortlich. Lambda ist dafür verantwortlich, Laufzeitaktualisierungen anzuwenden, wenn Sie diese Aktualisierungen vornehmen. Wenn Sie Ihre Funktion nicht aktualisieren, aktualisiert Lambda die Laufzeit nicht. Wenn Sie Ihre Funktion nicht regelmäßig aktualisieren, empfehlen wir dringend, diese für automatische Laufzeitaktualisierungen zu konfigurieren, damit sie weiterhin Sicherheitsupdates erhält.

Bei Funktionen, die für den Laufzeitaktualisierungsmodus Manual (Manuell) konfiguriert sind, sind Sie dafür verantwortlich, Ihre Funktion zu aktualisieren, damit sie die neueste Laufzeitversion verwendet. Wir empfehlen dringend, diesen Modus nur zu verwenden, um im seltenen Ereignis einer Inkompatibilität der Laufzeitaktualisierung die Laufzeitversion vorübergehend zurückzusetzen. Wir

empfehlen Ihnen außerdem, so schnell wie möglich in den Modus Auto (Automatisch) zu wechseln, um die Zeit zu minimieren, in der Ihre Funktionen nicht gepatcht werden.

Wenn Sie Container-Images zum Bereitstellen Ihrer Funktionen verwenden, ist Lambda für das Veröffentlichen aktualisierter Basis-Images verantwortlich. In diesem Fall sind Sie dafür verantwortlich, das Container-Image Ihrer Funktion aus dem neuesten Basis-Image neu zu erstellen und das Container-Image erneut bereitzustellen.

Dies ist in der folgenden Tabelle zusammengefasst:

Bereitstellungsmodus	Verantwortung von Lambda	Verantwortung des Kunden
Verwaltete Laufzeit, Modus Auto (Automatisch)	Veröffentlichen Sie neue Laufzeitversionen mit den neuesten Patches. Wenden Sie Laufzeit-Patches auf vorhandene Funktionen an.	Führen Sie im seltenen Fall eines Kompatibilitätsproblems bei Laufzeitaktualisierungen ein Rollback auf eine frühere Laufzeitversion durch. Folgen Sie den bewährten Methoden für Abwärtskompatibilität.
Verwaltete Laufzeit, Modus Function update (Funktionsaktualisierung)	Veröffentlichen Sie neue Laufzeitversionen mit den neuesten Patches.	Aktualisieren Sie die Funktionen regelmäßig, um die neueste Laufzeitversion zu erhalten. Schalten Sie eine Funktion in den Modus Auto (Automatisch), wenn Sie die Funktion nicht regelmäßig aktualisieren. Führen Sie im seltenen Fall eines Kompatibilitätsproblems bei Laufzeitaktualisierungen ein Rollback auf eine frühere Laufzeitversion durch. Folgen Sie den bewährten Methoden für Abwärtskompatibilität.
Verwaltete Laufzeit, Modus Manual (Manuell)	Veröffentlichen Sie neue Laufzeitversionen mit den neuesten Patches.	Verwenden Sie diesen Modus nur für ein vorübergehendes Laufzeit-Rollback im seltenen Fall eines Kompatibilitätsproblems bei der Aktualisierung.

Bereitstellungsmodus	Verantwortung von Lambda	Verantwortung des Kunden
		Schalten Sie die Funktionen so schnell wie möglich in den Modus Auto (Automatisch) oder Function update (Funktionsaktualisierung) und auf die neueste Laufzeitversion.
Container-Image	Veröffentlichen Sie neue Container-Images mit den neuesten Patches.	Stellen Sie die Funktionen regelmäßig neu bereit, indem Sie das neueste Container-Basis-Image verwenden, um die neuesten Patches abzurufen.

Weitere Informationen zur geteilten Verantwortung mit AWS finden Sie unter Modell der gemeinsamen Verantwortung.

Kontrolle der Lambda-Laufzeitaktualisierungsberechtigungen für Anwendungen mit hoher Konformität

Um Patching-Anforderungen zu erfüllen, verlassen sich Lambda-Kunden in der Regel auf automatische Laufzeitaktualisierungen. Wenn Ihre Anwendung strengen Anforderungen an die Aktualität von Patches unterliegt, sollten Sie die Verwendung früherer Runtime-Versionen einschränken. Sie können die Runtime-Management-Steuerungen von Lambda einschränken, indem Sie AWS Identity and Access Management (IAM) verwenden, um Benutzern in Ihrem AWS Konto den Zugriff auf den PutRuntimeManagementConfigAPI-Vorgang zu verweigern. Dieser Vorgang wird verwendet, um den Runtime-Aktualisierungsmodus für eine Event-Funktion auszuwählen.

Wenn Sie den Zugriff auf diesen Vorgang verweigern, werden alle Funktionen standardmäßig auf den Modus Auto (Automatisch) umgestellt. Sie können diese Einschränkung unternehmensweit anwenden, indem Sie Service-Kontrollrichtlinien (SCP) verwenden. Wenn Sie eine Funktion auf eine frühere Laufzeitversion zurücksetzen müssen, können Sie von Fall zu Fall eine Richtlinienausnahme gewähren.

Rufen Sie Daten über Lambda-Ereignisfunktionen ab, die eine veraltete Laufzeit verwenden

Wenn sich eine Lambda-Laufzeit dem Ende nähert, werden Sie von Lambda per E-Mail benachrichtigt und im Laufe der Zeit benachrichtigt. Health Dashboard Trusted Advisor In diesen E-Mails und Benachrichtigungen werden die $LATEST-Versionen von Funktionen aufgelistet, die die Laufzeit verwenden. Um alle Versionen Ihrer Event Function aufzulisten, die eine bestimmte Laufzeit verwenden, können Sie das AWS Command Line Interface (AWS CLI) oder eines der SDKs verwenden. AWS

Wenn Sie über eine große Anzahl von Funktionen verfügen, die eine Laufzeit verwenden, die bald nicht mehr unterstützt wird, können Sie auch die AWS SDKs AWS CLI oder verwenden, um Aktualisierungen Ihrer am häufigsten aufgerufenen Funktionen zu priorisieren.

In den folgenden Abschnitten erfahren Sie, wie Sie mithilfe der AWS SDKs AWS CLI und Daten über Ereignisfunktionen sammeln, die eine bestimmte Laufzeit verwenden.

Auflisten von Funktionsversionen, die eine bestimmte Laufzeit verwenden

Führen Sie den folgenden AWS CLI Befehl aus, um alle Versionen Ihrer Lambda-Ereignisfunktionen aufzulisten, die eine bestimmte Laufzeit verwenden. Dieser Befehl gibt keine Lambda Web Functions zurück. `RUNTIME_IDENTIFIER`Ersetzen Sie es durch den Namen der Laufzeit, die veraltet ist, und wählen Sie Ihren eigenen aus. AWS-Region Um nur die $LATEST-Funktionsversionen aufzulisten, lassen Sie `--function-version ALL` im Befehl weg.

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> Der Beispielbefehl listet Funktionen in der `us-east-1` Region für eine bestimmte Region auf. AWS-Konto Sie müssen diesen Befehl für jede Region wiederholen, in der Ihr Konto Funktionen hat, und für jede Ihrer Regionen. AWS-Konten

Mit einem der AWS SDKs können Sie auch Ereignisfunktionen auflisten, die eine bestimmte Laufzeit verwenden. Der folgende Beispielcode verwendet die V3 AWS SDK für JavaScript und die AWS SDK für Python (Boto3) , um eine Liste der Funktions-ARNs für Funktionen, die eine bestimmte Laufzeit

verwenden, zurückzugeben. Der Beispielcode gibt auch die CloudWatch Protokollgruppe für jede der aufgelisteten Funktionen zurück. Sie können diese Protokollgruppe verwenden, um das Datum des letzten Aufrufs der Funktion zu ermitteln. Weitere Informationen finden Sie im folgenden Abschnitt the section called “Identifizieren der am häufigsten und zuletzt aufgerufenen Funktionen”.

Node.js

Example JavaScript Code zum Auflisten von Funktionen, die eine bestimmte Laufzeit verwenden

```
import { LambdaClient, ListFunctionsCommand } from "@aws-sdk/client-lambda";
const lambdaClient = new LambdaClient();

const command = new ListFunctionsCommand({
    FunctionVersion: "ALL",
    MaxItems: 50
});
const response = await lambdaClient.send(command);

for (const f of response.Functions){
    if (f.Runtime == '<your_runtime>'){ // Use the runtime id, e.g. 'nodejs24.x' or
 'python3.14'
        console.log(f.FunctionArn);
        // get the CloudWatch log group of the function to
        // use later for finding the last invocation date
        console.log(f.LoggingConfig.LogGroup);
    }
}
// If your account has more functions than the specified
// MaxItems, use the returned pagination token in the
// next request with the 'Marker' parameter
if ('NextMarker' in response){
    let paginationToken = response.NextMarker;
  }
```

Python

Example Python-Code zur Auflistung von Funktionen mit einer bestimmten Laufzeit

```
import boto3
from botocore.exceptions import ClientError

def list_lambda_functions(target_runtime):
```

```
    lambda_client = boto3.client('lambda')

    response = lambda_client.list_functions(
        FunctionVersion='ALL',
        MaxItems=50
    )
    if not response['Functions']:
            print("No Lambda functions found")
    else:
        for function in response['Functions']:
            if function['PackageType']=='Zip' and function['Runtime'] ==
 target_runtime:
                print(function['FunctionArn'])
                # Print the CloudWatch log group of the function
                # to use later for finding last invocation date
                print(function['LoggingConfig']['LogGroup'])

    if 'NextMarker' in response:
       pagination_token = response['NextMarker']

if __name__ == "__main__":
    # Replace python3.12 with the appropriate runtime ID for your Lambda functions
    list_lambda_functions('python3.12')
```

Weitere Informationen zur Verwendung eines AWS SDK zum Auflisten Ihrer Funktionen mithilfe der ListFunctionsAktion finden Sie in der SDK-Dokumentation für Ihre bevorzugte Programmiersprache.

Sie können auch die Funktion AWS Config Erweiterte Abfragen verwenden, um alle Ihre Lambda-Ereignisfunktionen aufzulisten, die eine betroffene Laufzeit verwenden. AWS Config gibt keine Lambda Web Functions zurück. Diese Abfrage gibt nur die Versionen der Funktion $LATEST zurück, aber Sie können Abfragen aggregieren, um Funktionen für alle Regionen und mehrere Regionen AWS-Konten mit einem einzigen Befehl aufzulisten. Weitere Informationen finden Sie unter Abfragen des aktuellen Konfigurationsstatus von AWS Auto Scaling Ressourcen im AWS Config Entwicklerhandbuch.

Identifizieren der am häufigsten und zuletzt aufgerufenen Funktionen

Wenn Ihr System Funktionen AWS-Konto enthält, die eine Laufzeit verwenden, die bald veraltet sein wird, sollten Sie der Aktualisierung von Funktionen, die häufig aufgerufen werden, oder Funktionen, die kürzlich aufgerufen wurden, Priorität einräumen.

Wenn Sie nur über wenige Funktionen verfügen, können Sie die CloudWatch Logs-Konsole verwenden, um diese Informationen zu sammeln, indem Sie sich die Log-Streams Ihrer Funktionen ansehen. Weitere Informationen finden Sie unter An Logs gesendete CloudWatch Protokolldaten anzeigen.

Um die Anzahl der letzten Funktionsaufrufen zu sehen, können Sie auch die in der Lambda-Konsole angezeigten CloudWatch Metrikinformationen verwenden. Gehen Sie wie folgt vor, um diese Informationen anzuzeigen:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie eine Aufrufstatistik anzeigen möchten.
3. Wählen Sie den Tab Überwachung.
4. Legen Sie den Zeitraum fest, für den Sie die Statistiken anzeigen möchten, indem Sie den Datumsbereich auswählen. Die letzten Aufrufe werden im Bereich Aufrufe angezeigt.

Bei Konten mit einer größeren Anzahl von Funktionen kann es effizienter sein, diese Daten mithilfe der API-Aktionen AWS CLI oder eines der AWS SDKs programmgesteuert zu sammeln. DescribeLogStreamsGetMetricStatistics

Die folgenden Beispiele enthalten Codefragmente, die V3 AWS SDK für JavaScript und die verwenden, AWS SDK für Python (Boto3) um das Datum des letzten Aufrufs für eine bestimmte Funktion zu identifizieren und die Anzahl der Aufrufe für eine bestimmte Funktion in den letzten 14 Tagen zu bestimmen.

Node.js

Example JavaScript Code, um die Uhrzeit des letzten Aufrufs für eine Funktion zu ermitteln

```
import { CloudWatchLogsClient, DescribeLogStreamsCommand } from "@aws-sdk/client-
cloudwatch-logs";
const cloudWatchLogsClient = new CloudWatchLogsClient();
const command = new DescribeLogStreamsCommand({
    logGroupName: '<your_log_group_name>',
    orderBy: 'LastEventTime',
    descending: true,
    limit: 1
});
try {
    const response = await cloudWatchLogsClient.send(command);
    const lastEventTimestamp = response.logStreams.length > 0 ?
```

```
        response.logStreams[0].lastEventTimestamp : null;
    // Convert the UNIX timestamp to a human-readable format for display
    const date = new Date(lastEventTimestamp).toLocaleDateString();
    const time = new Date(lastEventTimestamp).toLocaleTimeString();
    console.log(`${date} ${time}`);

} catch (e){
    console.error('Log group not found.')
}
```

Python

Example Python-Code, um den Zeitpunkt des letzten Aufrufs für eine Funktion zu ermitteln

```
import boto3
from datetime import datetime

cloudwatch_logs_client  = boto3.client('logs')

response = cloudwatch_logs_client.describe_log_streams(
    logGroupName='<your_log_group_name>',
    orderBy='LastEventTime',
    descending=True,
    limit=1
)

try:
    if len(response['logStreams']) > 0:
        last_event_timestamp = response['logStreams'][0]['lastEventTimestamp']
        print(datetime.fromtimestamp(last_event_timestamp/1000)) # Convert timestamp
 from ms to seconds
    else:
        last_event_timestamp = None
except:
    print('Log group not found')
```

> **Tip**
>
> Sie können den Namen der Protokollgruppe Ihrer Funktion mithilfe der ListFunctionsAPI-Operation ermitteln. Der Code in the section called “Auflisten von Funktionsversionen, die eine bestimmte Laufzeit verwenden” enthält ein Beispiel für diese Vorgehensweise.

Node.js

Example JavaScript Code, um die Anzahl der Aufrufe in den letzten 14 Tagen zu ermitteln

```
import { CloudWatchClient, GetMetricStatisticsCommand } from "@aws-sdk/client-
cloudwatch";
const cloudWatchClient = new CloudWatchClient();
const command = new GetMetricStatisticsCommand({
    Namespace: 'AWS/Lambda',
    MetricName: 'Invocations',
    StartTime: new Date(Date.now()-86400*1000*14), // 14 days ago
    EndTime: new Date(Date.now()),
    Period: 86400 * 14, // 14 days.
    Statistics: ['Sum'],
    Dimensions: [{
        Name: 'FunctionName',
        Value: '<your_function_name>'
    }]
});
const response = await cloudWatchClient.send(command);
const invokesInLast14Days = response.Datapoints.length > 0 ?
    response.Datapoints[0].Sum : 0;

console.log('Number of invocations: ' + invokesInLast14Days);
```

Python

Example Python-Code, um die Anzahl der Aufrufe in den letzten 14 Tagen zu ermitteln

```
import boto3
from datetime import datetime, timedelta

cloudwatch_client = boto3.client('cloudwatch')

response = cloudwatch_client.get_metric_statistics(
    Namespace='AWS/Lambda',
    MetricName='Invocations',
    Dimensions=[
        {
            'Name': 'FunctionName',
            'Value': '<your_function_name>'
        },
    ],
```

```
    StartTime=datetime.now() - timedelta(days=14),
    EndTime=datetime.now(),
    Period=86400 * 14, # 14 days
    Statistics=[
        'Sum'
    ]
)

if len(response['Datapoints']) > 0:
    invokes_in_last_14_days = int(response['Datapoints'][0]['Sum'])
else:
    invokes_in_last_14_days = 0

print(f'Number of invocations: {invokes_in_last_14_days}')
```

Ändern der Laufzeitumgebung

Sie können interne Erweiterungen verwenden, um den Laufzeitprozess zu ändern. Interne Erweiterungen sind keine separaten Prozesse – sie werden als Teil des Laufzeitprozesses ausgeführt.

Lambda bietet sprachspezifische Umgebungsvariablen, die Sie festlegen können, um Optionen und Werkzeuge zur Laufzeit hinzuzufügen. Lambda bietet auch Wrapper-Skripts, die Lambda erlauben, den Laufzeit-Startup an Ihr Skript zu delegieren. Sie können ein Wrapper-Skript erstellen, um das Laufzeit-Startup-Verhalten anzupassen.

Language-specific Umgebungsvariablen

Lambda unterstützt Konfigurationsmöglichkeiten, mit denen Code während der Funktionsinitialisierung über die folgenden sprachspezifischen Umgebungsvariablen vorgeladen werden kann:

- `JAVA_TOOL_OPTIONS` – Unter Java unterstützt Lambda diese Umgebungsvariable, um zusätzliche Befehlszeilenvariablen in Lambda zu setzen. Mit dieser Umgebungsvariablen können Sie die Initialisierung von Werkzeugen angeben, insbesondere das Starten von nativen oder Java-Agenten mit den Optionen `agentlib` oder `javaagent`. Weitere Informationen finden Sie unter `JAVA_TOOL_OPTIONS`-Umgebungsvariablen.
- `NODE_OPTIONS`— In Node.js Laufzeiten verfügbar.
- `DOTNET_STARTUP_HOOKS` – Auf .NET Core 3.1 und höher gibt diese Umgebungsvariable einen Pfad zu einer Assembly (DLL) an, die Lambda verwenden kann.

Die Verwendung sprachspezifischer Umgebungsvariablen ist die bevorzugte Methode zum Festlegen von Startup-Eigenschaften.

Wrapper-Skripte

Sie können ein Wrapper-Skript erstellen, um das Laufzeitstartverhalten Ihrer Lambda-Ereignisfunktion anzupassen. Mit einem Wrapper-Skript können Sie Konfigurationsparameter festlegen, die nicht über sprachspezifische Umgebungsvariablen festgelegt werden können.

> **Note**
>
> Aufrufe können fehlschlagen, wenn das Wrapper-Skript den Laufzeitprozess nicht erfolgreich startet.

Wrapper-Skripts werden auf allen nativen Lambda-Laufzeiten unterstützt. Wrapper-Skripts werden auf OS-only Laufzeiten (der `provided`-Laufzeitfamilie) nicht unterstützt.

Wenn Sie ein Wrapper-Skript für Ihre Funktion verwenden, startet Lambda die Laufzeit mit Ihrem Skript. Lambda sendet den Pfad an Ihr Skript zum Interpreter und alle ursprünglichen Argumente für den Startup der Standardlaufzeit. Ihr Skript kann das Startup-Verhalten des Programms erweitern oder transformieren. Beispielsweise kann das Skript Argumente injizieren und ändern, Umgebungsvariablen festlegen oder Metriken, Fehler und andere Diagnoseinformationen erfassen.

Sie geben das Skript an, indem Sie den Wert der `AWS_LAMBDA_EXEC_WRAPPER`-Umgebungsvariablen als Dateisystempfad einer ausführbaren Binärdatei oder eines Skripts festlegen.

Beispiel: Erstellen und Verwenden eines Wrapper-Skripts als Lambda-Ebene

Im folgenden Beispiel erstellen Sie ein Wrapper-Skript, um den Python-Interpreter mit der `-X importtime`-Option zu starten. Wenn Sie die Funktion ausführen, generiert Lambda einen Protokolleintrag, der die Dauer der Importzeit für jeden Import anzeigt.

So erstellen und verwenden Sie ein Wrapper-Skript als Ebene

1. Erstellen Sie ein Verzeichnis für die Ebene:

   ```
   mkdir -p python-wrapper-layer/bin
   cd python-wrapper-layer/bin
   ```

2. Fügen Sie im Verzeichnis `bin` den folgenden Code in eine neue Datei namens `importtime_wrapper` ein. Dies ist das Wrapper-Skript.

   ```
   #!/bin/bash

   # the path to the interpreter and all of the originally intended arguments
   args=("$@")

   # the extra options to pass to the interpreter
   ```

```
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

3. Geben Sie dem Skript die Berechtigung zur Ausführung:

```
chmod +x importtime_wrapper
```

4. Erstellen Sie eine ZIP-Datei für die Ebene:

```
cd ..
zip -r ../python-wrapper-layer.zip .
```

5. Vergewissern Sie sich, dass Ihre .zip-Datei die folgende Verzeichnisstruktur aufweist:

```
python-wrapper-layer.zip
# bin
    # importtime_wrapper
```

6. Erstellen Sie eine Ebene mit dem .zip-Paket.
7. Erstellen Sie mit der Lambda-Konsole eine Event-Funktion.
   a. Öffnen Sie die Lambda-Konsole.
   b. Wählen Sie „Event-Funktion erstellen“.
   c. Geben Sie einen Function name (Funktionsnamen) ein.
   d. Wählen Sie für Laufzeit die neueste unterstützte Python-Laufzeit aus.
   e. Wählen Sie „Event-Funktion erstellen“.
8. Fügen Sie die Ebene zu Ihrer Funktion hinzu.
   a. Wählen Sie Ihre Funktion aus, und wählen Sie dann die Registerkarte Code, falls sie nicht bereits ausgewählt ist.
   b. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
   c. Wählen Sie unter Ebenenquelle die Option Benutzerdefinierte Ebenen und wählen Sie dann Ihre Ebene aus der Dropdown-Liste Benutzerdefinierte Ebenen aus.

d. Wählen Sie für Version die Option 1.

e. Wählen Sie Hinzufügen aus.

9. Fügen Sie die Wrapper-Umgebungsvariable hinzu.

   a. Wählen Sie den Tab Configuration (Konfiguration) und dann Environment variables (Umgebungsvariablen) aus.

   b. Wählen Sie unter Environment variables (Umgebungsvariablen) die Option Edit (Bearbeiten).

   c. Wählen Sie Umgebungsvariablen hinzufügen aus.

   d. Geben Sie für Key (Schlüssel) `AWS_LAMBDA_EXEC_WRAPPER` ein.

   e. Geben Sie als Wert `/opt/bin/importtime_wrapper` (`/opt/`+ die Ordnerstruktur Ihrer ZIP-Ebene) ein.

   f. Wählen Sie Speichern.

10. Testen Sie das Wrapper-Skript.

    a. Wählen Sie die Registerkarte Test.

    b. Wählen Sie unter Testereignis die Option Test aus. Sie müssen kein Testereignis erstellen — das Standardereignis funktioniert.

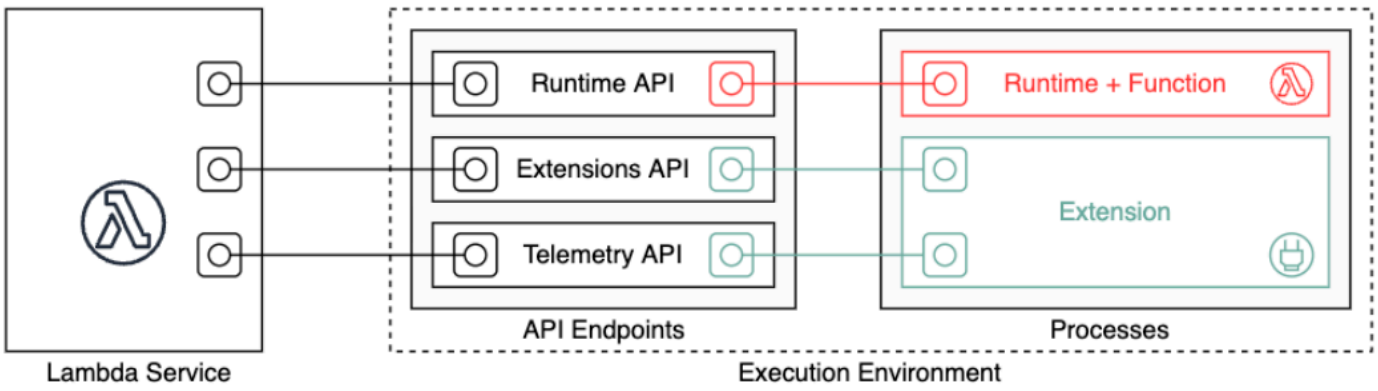
    c. Scrollen Sie nach unten zu Protokollausgabe. Da Ihr Wrapper-Skript den Python-Interpreter mit der `-X importtime`-Option gestartet hat, zeigen die Protokolle die für jeden Import erforderliche Zeit an. Zum Beispiel:

```
532 |           collections
import time:        63 |         63 |           _functools
import time:      1053 |       3646 |         functools
import time:      2163 |       7499 |       enum
import time:       100 |        100 |         _sre
import time:       446 |        446 |           re._constants
import time:       691 |       1136 |         re._parser
import time:       378 |        378 |         re._casefix
import time:       670 |       2283 |       re._compiler
import time:       416 |        416 |       copyreg
```

Verwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten

AWS Lambda bietet eine HTTP-API für benutzerdefinierte Laufzeiten von Lambda-Ereignisfunktionen, um Aufrufereignisse von Lambda zu empfangen und Antwortdaten innerhalb der Lambda-Ausführungsumgebung zurückzusenden. Dieser Abschnitt enthält die API-Referenz für die Lambda-Laufzeiten-API.

> **Lambda Managed Instances unterstützen gleichzeitige Anfragen**
>
> Lambda Managed Instances verwenden dieselbe Runtime-API wie Funktionen, die den Lambda-Standardkapazitätsanbieter verwenden. Der Hauptunterschied besteht darin, dass verwaltete Instances gleichzeitige `/response` Anfragen `/next` und Anfragen bis zum konfigurierten Limit akzeptieren können. `AWS_LAMBDA_MAX_CONCURRENCY` Dadurch können mehrere Aufrufe gleichzeitig in einer einzigen Ausführungsumgebung verarbeitet werden. Weitere Informationen zu verwalteten Instanzen finden Sie unter. Grundlegendes zur Lambda Managed Instance-Ausführungsumgebung



Die OpenAPI-Spezifikation für die Laufzeit-API-Version 2018-06-01 ist unter runtime-api.zip verfügbar.

Um eine API-Anforderungs-URL zu erstellen, rufen Laufzeitumgebungen den API-Endpunkt aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariablen ab, fügen die API-Version und dann den gewünschten Ressourcenpfad hinzu.

Example Anforderung

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API-Methoden

- Nächster Aufruf
- Aufrufantwort
- Initialisierungsfehler
- Aufruffehler
- After-Restore (gilt nur für SnapStart)
- Wiederherstellungsfehler (gilt nur für SnapStart)

Nächster Aufruf

Pfad – `/runtime/invocation/next`

Methode – GET

Die Laufzeit sendet diese Meldung an Lambda, um ein Aufrufereignis anzufordern. Der Antworttext enthält die Nutzlast aus dem Aufruf. Dabei handelt es sich um ein JSON-Dokument, das Ereignisdaten aus dem Funktionsauslöser enthält. Die Antwort-Header enthalten zusätzliche Daten zum Aufruf.

Antwort-Header

- `Lambda-Runtime-Aws-Request-Id`— Das Ereignis, das den Funktionsaufruf ausgelöst hat. Ereignisquellen stellen Anforderungs-IDs bereit, oder Lambda generiert sie bei der Aufnahme automatisch. Eine einzelne Anforderungs-ID kann zu mehreren Aufrufversuchen führen. Verwenden Sie es im URL-Pfad, wenn Sie die Antwort oder den Fehler senden.

  Beispiel, `8476a536-e9f4-11e8-9739-2dfe598c3fcd`.
- `Lambda-Runtime-Deadline-Ms` – Das Datum, an dem eine Zeitüberschreitung für die Funktion eintritt (in Unix-Millisekunden).

  Beispiel, `1542409706888`.
- `Lambda-Runtime-Invoked-Function-Arn` – Der ARN der/des Lambda-Funktion, -Version oder -Alias, die/der im Aufruf angegeben ist.

  Beispiel, `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`.
- `Lambda-Runtime-Trace-Id` – Der AWS X-Ray -Nachverfolgungs-Header.

Beispiel, `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`.

- `Lambda-Runtime-Client-Context`— Bei Aufrufen aus dem AWS Mobile SDK Daten über die Client-Anwendung und das Gerät.
- `Lambda-Runtime-Cognito-Identity`— Für Aufrufe aus dem AWS Mobile SDK Daten über den Amazon Cognito Cognito-Identitätsanbieter.
- `Lambda-Runtime-Invocation-Id`— Eine eindeutige Kennung für diesen Aufrufversuch.

Legen Sie kein Timeout für die `GET`-Anfrage fest, da sich die Antwort verzögern könnte. Zwischen dem Zeitpunkt, an dem Lambda die Laufzeit bootet, und dem Zeitpunkt, an dem die Laufzeit ein Ereignis zurückgibt, kann der Laufzeitprozess für mehrere Sekunden eingefroren werden.

Eine Anforderungs-ID (`Lambda-Runtime-Aws-Request-Id`) identifiziert ein eindeutiges Ereignis. Anforderungs-IDs werden von Ereignisquellen bereitgestellt oder bei der Aufnahme automatisch von Lambda generiert. Verwenden Sie die Anforderungs-ID im URL-Pfad, wenn Sie die Antwort oder den Fehler senden.

Eine Aufruf-ID (`Lambda-Runtime-Invocation-Id`) steht für einen einzelnen Aufrufversuch für ein Ereignis. Eine einzelne Anforderungs-ID kann zu mehreren Aufrufversuchen führen, von denen jeder seine eigene eindeutige Aufruf-ID hat. Lambda verwendet jede Aufruf-ID genau einmal und verwendet sie nie wieder. Geben Sie diesen Wert wieder ein und rufen Sie auf. `/response` `/error` Der Header ist aus Gründen der Abwärtskompatibilität mit bestehenden Laufzeiten optional. Wenn Sie ihn weglassen, wird er nicht zurückgewiesen. Lambda lehnt nur mit ab, `400` `InvalidInvocationId` wenn der Header vorhanden ist, sein Wert jedoch nicht mit dem aktiven Aufruf übereinstimmt.

Der Nachverfolgungs-Header enthält die Nachverfolgungs-ID, die ID des übergeordneten Segments und die Erfassungsentscheidung Wenn die Anforderung erfasst wurde, wurde die Anforderung von Lambda oder einem Upstream-Service erfasst. Die Laufzeit sollte die `_X_AMZN_TRACE_ID` auf den Wert des Headers festlegen. Das X-Ray SDK liest dies, um die IDs abzurufen und zu bestimmen, ob die Anfrage verfolgt werden soll.

Aufrufantwort

Pfad – `/runtime/invocation/`*`AwsRequestId`*`/response`

Methode – POST

Nachdem die Funktion bis zum Abschluss ausgeführt wurde, schickt die Laufzeitumgebung eine Aufrufantwort an Lambda. Im Fall synchroner Aufrufe schickt Lambda die Antwort anschließend an den Client zurück.

Anfordern von Headern

`Lambda-Runtime-Invocation-Id`— Gibt den empfangenen Wert von zurück`/next`. Lambda lehnt die Anfrage mit ab, `400 InvalidInvocationId` wenn der Wert nicht mit dem aktiven Aufruf übereinstimmt.

Example Erfolg für Anforderung

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
INVOCATION_ID=<value from Lambda-Runtime-Invocation-Id response header>
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS" --header "Lambda-Runtime-Invocation-Id: $INVOCATION_ID"
```

Initialisierungsfehler

Wenn die Funktion einen Fehler zurückgibt oder die Laufzeit während der Initialisierung auf einen Fehler stößt, verwendet die Laufzeit diese Methode, um den Fehler an Lambda zu melden.

Pfad – `/runtime/init/error`

Methode – POST

Header

`Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Dieser Header ist optional. Lambda akzeptiert jeden Zeichenkettenwert. Wir empfehlen, das Format zu verwenden`<Category.Reason>`, in dem Category für `Runtime` oder steht `Function` und Reason mit einem Großbuchstaben beginnt. Beispiel:

- `Runtime.NoSuchHandler`
- `Runtime.APIKeyNotFound`
- `Runtime.ConfigInvalid`
- `Runtime.BeforeSnapshotError`(für) SnapStart
- `Runtime.UnknownReason`

Werte, die diesem Muster nicht entsprechen, werden auf `Runtime.Unknown` oder `Function.Unknown` normalisiert.

Body-Parameter

`ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antworttextparameter:

- `StatusResponse` – Zeichenfolge. Statusinformationen, gesendet mit 202 Antwortcodes.
- `ErrorResponse`— Zusätzliche Fehlerinformationen, die zusammen mit den Fehlerantwortcodes gesendet werden. ErrorResponse enthält einen Fehlertyp und eine Fehlermeldung.

Antwortcodes

- 202 – Akzeptiert
- 403 – Verboten
- 500 — Container-Fehler. Non-recoverable Bundesstaat. Die Laufzeit sollte umgehend beendet werden.

Example Initialisierungsfehleranforderung

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Aufruffehler

Wenn die Funktion einen Fehler zurückgibt oder die Laufzeit auf einen Fehler stößt, verwendet die Laufzeitumgebung diese Methode, um den Fehler an Lambda zu melden.

Pfad – `/runtime/invocation/`*`AwsRequestId`*`/error`

Methode – POST

Header

`Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Erforderlich: Nein.

Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

`Lambda-Runtime-Invocation-Id`— Gibt den empfangenen Wert von zurück`/next`. Lambda lehnt die Anfrage mit ab, `400 InvalidInvocationId` wenn der Wert nicht mit dem aktiven Aufruf übereinstimmt.

Body-Parameter

`ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
```

```
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antworttextparameter:

- `StatusResponse` – Zeichenfolge. Statusinformationen, gesendet mit 202 Antwortcodes.
- `ErrorResponse`— Zusätzliche Fehlerinformationen, die zusammen mit den Fehlerantwortcodes gesendet werden. ErrorResponse enthält einen Fehlertyp und eine Fehlermeldung.

Antwortcodes

- 202 – Akzeptiert
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 — Container-Fehler. Non-recoverable Bundesstaat. Die Laufzeit sollte umgehend beendet werden.

Example Fehler für Anforderung

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

After-Restore (gilt nur für SnapStart)

Pfad – `/runtime/restore/next`

Methode – GET

Nach Abschluss der Before-Snapshot-Hooks werden die Runtime-Aufrufe ausgeführt. `GET /runtime/restore/next` Dies ist ein Blockierungsaufruf im Iterator-Stil, ähnlich wie, der Lambda signalisiert`/runtime/invocation/next`, dass die Laufzeit bereit ist, um einen Snapshot der Ausführungsumgebung zu erstellen. Die Anfrage wird blockiert, bis Lambda die Ausführungsumgebung aus einem Snapshot wiederherstellt, und gibt dann eine HTTP 200-Antwort mit einem leeren Hauptteil zurück.

Header

Keine Header erforderlich.

Antwortcodes

- 200 — Lambda hat die Ausführungsumgebung wiederhergestellt. Führen Sie Hooks nach der Wiederherstellung aus. Der Antworttext ist leer.
- 403 — Verboten. Die Laufzeit befindet sich nicht in einem Zustand, der dies zulässt `/restore/next` (z. B. hat die Laufzeitumgebung bereits `/invocation/next` oder aufgerufen`/restore/next`).
- 404 — SnapStart ist für diese Funktion nicht aktiviert.
- 500 – Container-Fehler. Die Ausführungsumgebung befindet sich in einem Zustand, der nicht wiederhergestellt werden kann. Beenden Sie den Runtime-Prozess.

Erforderliche Syntax

```
GET /2018-06-01/runtime/restore/next HTTP/1.1
Host: ${AWS_LAMBDA_RUNTIME_API}
```

Antwortsyntax

```
HTTP/1.1 200 OK
Content-Length: 0
```

> **Note**
>
> Legen Sie für diese (oder eine andere) Runtime-API-Anfrage kein clientseitiges Socket- oder Lese-Timeout fest. Dies ist ein blockierender Aufruf im Iterator-Stil; Lambda friert die Ausführungsumgebung ein, während die Anfrage geöffnet ist. Die Anfrage kann für die gesamte Lebensdauer des Snapshots (möglicherweise Tage, Wochen oder länger) geöffnet bleiben, ohne dass die Verbindung vom Lambda-Service als inaktiv betrachtet wird.

Wiederherstellungsfehler (gilt nur für SnapStart)

Wenn ein Hook nach der Wiederherstellung fehlschlägt oder die Runtime während der Wiederherstellung auf einen Fehler stößt, verwendet die Runtime diese Methode, um den Fehler an Lambda zu melden. Lambda schlägt beim In-Flight-Aufruf fehl und zerstört die Ausführungsumgebung.

Pfad – `/runtime/restore/error`

Methode – POST

Header

`Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Dieser Header ist optional. Lambda akzeptiert jeden Zeichenkettenwert. Wir empfehlen die Verwendung des Formats`<Category.Reason>`, in dem Category für `Runtime` oder steht `Function` und Reason mit einem Großbuchstaben beginnt (z. B.`Runtime.AfterRestoreError`). Werte, die diesem Muster nicht entsprechen, werden auf oder normalisiert. `Runtime.Unknown` `Function.Unknown`

Antwortcodes

- 202 — Akzeptiert. Die Antwortstelle ist`{"status":"OK"}`. Die Laufzeit sollte den Prozess beenden.
- 403 — Verboten. Die Laufzeit befindet sich nicht in einem Zustand, der dies zulässt `/restore/error` (wurde z. B. nicht aufgerufen). `/restore/next`
- 404 — SnapStart ist für diese Funktion nicht aktiviert.
- 500 – Container-Fehler. Die Ausführungsumgebung befindet sich in einem Zustand, der nicht wiederhergestellt werden kann. Beenden Sie den Runtime-Prozess.

Erforderliche Syntax

```
POST /2018-06-01/runtime/restore/error HTTP/1.1
Host: ${AWS_LAMBDA_RUNTIME_API}
Lambda-Runtime-Function-Error-Type: Runtime.AfterRestoreError
```

Antwortsyntax

```
HTTP/1.1 202 Accepted
Content-Type: application/json

{"status":"OK"}
```

Wann sollten die Laufzeiten von Lambda verwendet werden OS-only

Lambda bietet verwaltete Laufzeiten für Java, Python Node.js, .NET und Ruby. Um Lambda-Ereignisfunktionen in einer Programmiersprache zu erstellen, die nicht als verwaltete Laufzeit verfügbar ist, verwenden Sie eine OS-only Runtime (die `provided` Runtime-Familie). Es gibt drei Hauptanwendungsfälle für OS-only Laufzeiten:

- Native Ahead-of-Time-Kompilierung (AOT): Sprachen wie Go, Rust, Swift und C++ werden nativ zu einer ausführbaren Binärdatei kompiliert, für die keine spezielle Sprachlaufzeit erforderlich ist. Diese Sprachen benötigen nur eine Betriebssystemumgebung, in der die kompilierte Binärdatei ausgeführt werden kann. Sie können OS-only Lambda-Laufzeiten auch verwenden, um Binärdateien bereitzustellen, die mit.NET Native AOT und Java GraalVM Native Image kompiliert wurden.

  Sie müssen einen Laufzeitschnitten-Client in Ihre Binärdatei aufnehmen. Der Laufzeitenschnittstellen-Client ruft die Verwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten auf, um Funktionsaufrufe abzurufen, und ruft dann den Funktionshandler auf. Lambda bietet Runtime-Schnittstellenclients für Rust, Go, .NET Native AOT, Swift (experimentell) und C++ (experimentell).

  Sie müssen Ihre Binärdatei für eine Linux-Umgebung und für dieselbe Befehlssatzarchitektur kompilieren, die Sie für die Funktion verwenden möchten (x86_64 oder arm64).
- Third-party Laufzeiten: Sie können Funktionen mit handelsüblichen Laufzeiten wie Bref für PHP ausführen.
- Benutzerdefinierte Laufzeiten: Sie können Ihre eigene Laufzeit für eine Sprache oder Sprachversion erstellen, für die Lambda keine verwaltete Laufzeit bereitstellt, z. B. 19. Node.js Weitere Informationen finden Sie unter Erstellen einer benutzerdefinierten Laufzeit für AWS Lambda Funktionen für Ereignisse. Dies ist der am wenigsten verbreitete Anwendungsfall für OS-only Laufzeiten.

Lambda unterstützt die folgenden OS-only Laufzeiten:

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
OS-only Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029

Die Laufzeit von Amazon Linux 2023 (`provided.al2023`) bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie `glibc`.

Die `provided.al2023`-Laufzeit verwendet `dnf` als Paketmanager anstelle von `yum`, was der Standard-Paketmanager in Amazon Linux 2 ist. Weitere Informationen zu den Unterschieden zwischen `provided.al2023` und `provided.al2` finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.

Erstellen einer benutzerdefinierten Laufzeit für AWS Lambda Funktionen für Ereignisse

Sie können eine AWS Lambda Runtime in jeder Programmiersprache implementieren. Eine Laufzeit ist ein Programm, das die Handler-Methode einer Lambda-Ereignisfunktion ausführt, wenn die Funktion aufgerufen wird. Die Laufzeit kann im Bereitstellungspaket Ihrer Funktion enthalten sein oder in einem Layer verteilt werden. Wenn Sie die Funktion erstellen, wählen Sie eine OS-only Laufzeit (die `provided` Runtime-Familie).

> **Note**
>
> Das Erstellen einer benutzerdefinierten Laufzeit ist ein Anwendungsfall für Fortgeschrittene. Informationen zur Kompilierung in eine native Binärdatei oder zur Verwendung einer Drittanbieter-Laufzeit finden Sie unter Wann sollten die Laufzeiten von Lambda verwendet werden OS-only.

Eine exemplarische Vorgehensweise für die Bereitstellung einer benutzerdefinierten Laufzeit finden Sie unter Tutorial: Erstellen einer benutzerdefinierten Laufzeit.

Themen

- Voraussetzungen
- Implementieren von Antwort-Streaming in einer benutzerdefinierten Laufzeitumgebung
- Erstellung benutzerdefinierter Laufzeiten für Lambda Managed Instances

Voraussetzungen

Benutzerdefinierte Laufzeiten müssen bestimmte Initialisierungs- und Verarbeitungsaufgaben abschließen. Eine Laufzeit führt den Setup-Code der Funktion aus, liest den Namen des Handlers

aus einer Umgebungsvariablen und liest Aufrufereignisse von der Lambda-Laufzeit-API. Die Laufzeit übergibt die Ereignisdaten an den Funktions-Handler und sendet die Antwort aus dem Handler zurück an Lambda.

Initialisierungsaufgaben

Die Initialisierungsaufgaben werden einmal pro Instance der Funktion ausgeführt, um die Umgebung auf die Verarbeitung von Aufrufen vorzubereiten.

- Einstellungen abrufen – Liest Umgebungsvariablen, um Details zu Funktion und Umgebung abzurufen.
  - `_HANDLER` – Der Speicherort für den Handler aus der Konfiguration der Funktion. Das Standardformat ist *`file`*`.`*`method`*, wobei `file` der Name der Datei ohne Erweiterung und `method` der Name einer Methode oder Funktion ist, die in der Datei definiert ist.
  - `LAMBDA_TASK_ROOT` – Das Verzeichnis, das den Funktionscode enthält.
  - `AWS_LAMBDA_RUNTIME_API` – Host und Port der Laufzeit-API.

  Unter Definierte Laufzeitumgebungsvariablen finden Sie eine vollständige Liste der verfügbaren Variablen.
- Funktion initialisieren – Lädt die Handler-Datei und führt den globalen oder statischen Code aus, den sie enthält. Funktionen sollten statische Ressourcen wie SDK-Clients und Datenbankverbindungen einmal erstellen und für mehrere Aufrufe wiederverwenden.
- Fehler verarbeiten – Wenn ein Fehler auftritt, werden die Initialisierungsfehler-API aufgerufen die Ausführung sofort beendet.

Die Initialisierung zählt zur fakturierten Ausführungszeit und zum Timeout. Wenn eine Ausführung die Initialisierung einer neuen Instance Ihrer Funktion auslöst, sehen Sie die Initialisierungszeit in den Protokollen und in der AWS X-Ray -Nachverfolgung.

Example log

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

Verarbeitungsaufgaben

Während der Ausführung verwendet eine Laufzeit die Lambda-Laufzeitschnittstelle, um eingehende Ereignisse zu verwalten und Fehler zu melden. Nach Abschluss der Initialisierungsaufgaben

verarbeitet die Laufzeit eingehende Ereignisse in einer Schleife. Führen Sie in Ihrem Laufzeitcode die folgenden Schritte in der angegebenen Reihenfolge aus.

- Ereignis abrufen – Ruft die API für den nächsten Aufruf auf, um das nächste Ereignis abzurufen. Der Antworttext enthält die Ereignisdaten. De Antwort-Header enthalten die Anforderungs-ID und andere Informationen.
- Den Tracing-Header propagieren — Ruft den X-Ray Tracing-Header aus dem `Lambda-Runtime-Trace-Id` Header in der API-Antwort ab. Legen Sie die `_X_AMZN_TRACE_ID`-Umgebungsvariable lokal mit demselben Wert fest. Das X-Ray SDK verwendet diesen Wert, um Trace-Daten zwischen Diensten zu verbinden.
- Context-Objekt erstellen – Erstellt ein Objekt mit Kontextinformationen aus Umgebungsvariablen und Headern in den API-Antworten.
- Funktions-Handler aufrufen – Übergibt Ereignis und Context-Objekt an den Handler.
- Antwort verarbeiten – Ruft die API für die Aufrufantwort ab, um die Antwort aus dem Handler zu veröffentlichen.
- Fehler verarbeiten – Ruft die API für Aufruffehler auf, wenn ein Fehler auftritt.
- Bereinigen Gibt nicht verwendete Ressourcen frei, sendet Daten an andere Services oder führt zusätzliche Aufgaben aus, bevor das nächste Ereignis abgerufen wird.

Eintrittspunkt

Der Eintrittspunkt einer benutzerdefinierten Laufzeit ist eine ausführbare Datei mit dem Namen `bootstrap`. Bei der Bootstrap-Datei kann es sich um die Laufzeit handeln oder es wird eine andere Datei aufgerufen, die die Laufzeit erstellt. Wenn das Stammverzeichnis Ihres Bereitstellungspakets keine Datei mit dem Namen `bootstrap` enthält, sucht Lambda in den Funktionsschichten nach der Datei. Wenn die `bootstrap`-Datei nicht existiert oder nicht ausgeführt werden kann, gibt die Funktion beim Aufruf einen `Runtime.InvalidEntrypoint`-Fehler zurück.

Hier ist eine `bootstrap` Beispieldatei, die eine gebündelte Version von verwendet Node.js , um eine JavaScript Runtime in einer separaten Datei mit dem Namen `runtime.js` auszuführen.

Example bootstrap

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

Implementieren von Antwort-Streaming in einer benutzerdefinierten Laufzeitumgebung

Für Antwort-Streaming-Funktionen haben die Endpunkte `response` und `error` ein leicht geändertes Verhalten, das es der Laufzeit ermöglicht, Teilantworten an den Client zu streamen und Nutzlasten in Paketen zurückzugeben. Weitere Informationen zu dem spezifischen Verhalten finden Sie unter:

- `/runtime/invocation/AwsRequestId/response` – Gibt den `Content-Type`-Header von der Laufzeit zum Senden an den Client weiter. Lambda gibt die Antwortnutzlast in Abschnitten zurück, indem es die Chunked Transfer Encoding verwendet HTTP/1.1 . Um die Antwort an Lambda zu streamen, muss die Laufzeit:
  - Den `Lambda-Runtime-Function-Response-Mode`-HTTP-Header auf `streaming` festlegen.
  - Legen Sie den `Transfer-Encoding`-Header auf `chunked` fest.
  - Schreiben Sie die Antwort, die der Spezifikation für die Verschlüsselung von Chunked Transfer entspricht. HTTP/1.1
  - Schließt die zugrunde liegende Verbindung, nachdem sie die Antwort erfolgreich geschrieben hat.
- `/runtime/invocation/AwsRequestId/error` – Die Laufzeit kann diesen Endpunkt verwenden, um Funktions- oder Laufzeitfehler an Lambda zu melden, das auch den `Transfer-Encoding`-Header akzeptiert. Dieser Endpunkt kann nur aufgerufen werden, bevor die Laufzeit mit dem Senden einer Aufrufantwort beginnt.
- Melden von Fehlern in der Mitte des Prozesses mit Hilfe von Fehler-Trailern in `/runtime/invocation/AwsRequestId/response` – Um Fehler zu melden, die auftreten, nachdem die Laufzeit mit dem Schreiben der Antwort auf den Aufruf begonnen hat, kann die Laufzeit optional HTTP-Trailing-Header namens `Lambda-Runtime-Function-Error-Type` und `Lambda-Runtime-Function-Error-Body` anfügen. Lambda behandelt dies als erfolgreiche Antwort und leitet die Fehler-Metadaten, die die Laufzeit bereitstellt, an den Client weiter.

  > **Note**
  >
  > Um nachgestellte `Trailer`-Header anzufügen, muss die Laufzeit den Header-Wert am Anfang der HTTP-Anfrage festlegen. Dies ist eine Anforderung der Spezifikation für die Verschlüsselung von HTTP/1.1 Chunked Transfer.

  - `Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert

jede Zeichenfolge, wir empfehlen jedoch ein Format von`<category.reason>`. Beispiel, `Runtime.APIKeyNotFound`.

- `Lambda-Runtime-Function-Error-Body`— Base64-encoded Informationen über den Fehler.

Erstellung benutzerdefinierter Laufzeiten für Lambda Managed Instances

Lambda Managed Instances verwenden dieselbe Runtime-API wie Funktionen, die den Lambda-Standardkapazitätsanbieter verwenden. Es gibt jedoch wesentliche Unterschiede in der Art und Weise, wie benutzerdefinierte Laufzeiten implementiert werden müssen, um das Modell der gleichzeitigen Ausführung von Managed Instances zu unterstützen.

Gleichzeitige Bearbeitung von Anfragen

Der Hauptunterschied bei der Erstellung benutzerdefinierter Laufzeiten für verwaltete Instanzen ist die Unterstützung gleichzeitiger Aufrufe. Im Gegensatz zu Funktionen, die den Lambda-Standardkapazitätsanbieter verwenden, bei denen die Laufzeit jeweils einen Aufruf verarbeitet, können Managed Instances mehrere Aufrufe gleichzeitig innerhalb einer einzigen Ausführungsumgebung verarbeiten.

Ihre benutzerdefinierte Laufzeit muss:

- Support gleichzeitiger **`/next`** Anfragen — Die Runtime kann mehrere gleichzeitige Aufrufe an die nächste Aufruf-API tätigen, bis zu dem in der `AWS_LAMBDA_MAX_CONCURRENCY` Umgebungsvariablen angegebenen Grenzwert.
- Gleichzeitige **`/response`** Anfragen verarbeiten — Bei mehreren Aufrufen kann die Aufruf-Antwort-API gleichzeitig aufgerufen werden.
- Implementieren Sie eine threadsichere Bearbeitung von Anfragen — Stellen Sie sicher, dass sich gleichzeitige Aufrufe nicht gegenseitig behindern, indem Sie gemeinsam genutzte Ressourcen und den Status ordnungsgemäß verwalten.
- Anforderungs-IDs und Aufruf-IDs korrekt verwenden — Eine Anforderungs-ID identifiziert das Ereignis; verwenden Sie sie im URL-Pfad. Eine Aufruf-ID steht für einen einzelnen Aufrufversuch — eine einzelne Anforderungs-ID kann zu mehreren Versuchen mit jeweils einer eindeutigen Aufruf-ID führen. Geben Sie die Aufruf-ID wieder ein und rufen Sie auf. `/response` `/error` Der Header ist optional. Aus Gründen der Abwärtskompatibilität wird es akzeptiert, ihn wegzulassen. Lambda lehnt nur mit ab`400 InvalidInvocationId`, wenn der Header vorhanden ist, sein Wert jedoch nicht übereinstimmt.

Implementierungsmuster

Ein typisches Implementierungsmuster für Laufzeiten von Managed Instances beinhaltet die Erstellung von Worker-Threads oder Prozessen zur Verarbeitung gleichzeitiger Aufrufe:

1. Das Parallelitätslimit auslesen — Lesen Sie bei der Initialisierung die `AWS_LAMBDA_MAX_CONCURRENCY` Umgebungsvariable, um zu bestimmen, wie viele gleichzeitige Aufrufe unterstützt werden sollen.
2. Worker-Pool erstellen — Initialisieren Sie einen Worker-Pool (Threads, Prozesse oder asynchrone Aufgaben), der dem Parallelitätslimit entspricht.
3. Worker-Verarbeitungsschleife — Jeder Worker unabhängig:
   - Ruft `/runtime/invocation/next` auf, um ein Aufrufereignis abzurufen
   - Ruft den Funktionshandler mit den Ereignisdaten auf
   - Sendet die Antwort an `/runtime/invocation/AwsRequestId/response`
   - Wiederholt die Schleife

Weitere Überlegungen

- Protokollierungsformat — Verwaltete Instances unterstützen nur das JSON-Protokollformat. Stellen Sie sicher, dass Ihre Laufzeit die `AWS_LAMBDA_LOG_FORMAT` Umgebungsvariable respektiert und nur das JSON-Format verwendet. Weitere Informationen finden Sie unter Konfiguration der JSON- und Klartext-Protokollformate.
- Gemeinsam genutzte Ressourcen — Seien Sie vorsichtig, wenn Sie gemeinsam genutzte Ressourcen wie das `/tmp` Verzeichnis mit gleichzeitigen Aufrufen verwenden. Implementieren Sie geeignete Verriegelungsmechanismen, um Rennbedingungen zu vermeiden.

Weitere Informationen zu Ausführungsumgebungen für Lambda Managed Instances finden Sie unterGrundlegendes zur Lambda Managed Instance-Ausführungsumgebung.

Tutorial: Erstellen einer benutzerdefinierten Laufzeit

In diesem Tutorial erstellen Sie eine Lambda-Ereignisfunktion mit einer benutzerdefinierten Laufzeit. Sie beginnen, indem Sie die Laufzeit in das Bereitstellungspaket der Funktion einfügen. Anschließend migrieren Sie sie zu einer Ebene, die Sie unabhängig von der Funktion verwalten. Schließlich geben Sie die Laufzeitebene frei, indem Sie ihre ressourcenbasierte Berechtigungsrichtlinie aktualisieren.

Voraussetzungen

In diesem Tutorial wird davon ausgegangen, dass Sie über Kenntnisse zu den grundlegenden Lambda-Operationen und der Lambda-Konsole verfügen. Sofern noch nicht geschehen, befolgen Sie die Anweisungen unter Erstellen einer Lambda-Funktion mit der Konsole, um Ihre erste Lambda-Funktion zu erstellen.

Um die folgenden Schritte durchzuführen, benötigen Sie die AWS CLI Version 2. Befehle und die erwartete Ausgabe werden in separaten Blöcken aufgeführt:

```
aws --version
```

Die Ausgabe sollte folgendermaßen aussehen:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

Bei langen Befehlen wird ein Escape-Zeichen (\) wird verwendet, um einen Befehl über mehrere Zeilen zu teilen.

Verwenden Sie auf Linux und macOS Ihren bevorzugten Shell- und Paket-Manager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux. Die CLI-Beispielbefehle in diesem Handbuch verwenden die Linux-Formatierung. Befehle, die Inline-JSON-Dokumente enthalten, müssen neu formatiert werden, wenn Sie die Windows-CLI verwenden.

Sie benötigen eine IAM-Rolle, um eine Lambda Funktion zu erstellen. Die Rolle benötigt die Erlaubnis, Protokolle an Logs zu CloudWatch senden und auf die Daten zuzugreifen AWS-Services , die Ihre Funktion verwendet. Wenn Sie nicht über eine Rolle für die Funktionserstellung verfügen, erstellen Sie jetzt eine.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.

2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.

   - Trusted entity (Vertrauenswürdige Entität) – Lambda.
   - Berechtigungen — AWSLambdaBasicExecutionRole.
   - Role name (Name der Rolle – **`lambda-role`**.

   Die AWSLambdaBasicExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion benötigt, um Protokolle in Logs zu CloudWatch schreiben.

Erstellen einer -Funktion

Erstellen Sie eine Lambda-Ereignisfunktion mit einer benutzerdefinierten Laufzeit. Dieses Beispiel enthält zwei Dateien: eine `bootstrap`-Laufzeitdatei und einen Funktions-Handler. Beide sind in Bash implementiert.

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir runtime-tutorial
   cd runtime-tutorial
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `bootstrap`. Dies ist die benutzerdefinierte Laufzeit.

   Example bootstrap

   ```
   #!/bin/sh

   set -euo pipefail

   # Initialization - load function handler
   source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

   # Processing
   while true
   do
     HEADERS="$(mktemp)"
     # Get an event. The HTTP request will block until one is received
     EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
   ${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")
   ```

```
  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Extract invocation ID for cross-wiring protection
  INVOCATION_ID=$(grep -Fi Lambda-Runtime-Invocation-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response, echoing the invocation ID
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE" \
    ${INVOCATION_ID:+--header "Lambda-Runtime-Invocation-Id: $INVOCATION_ID"}
done
```

Die Laufzeit lädt ein Funktionsskript aus dem Bereitstellungspaket. Sie verwendet zwei Variablen, um das Skript zu finden. `LAMBDA_TASK_ROOT` gibt an, wohin das Paket extrahiert wurde, und `_HANDLER` enthält den Namen des Skripts.

Nachdem die Laufzeit das Funktionsskript geladen hat, verwendet sie die Laufzeit-API, um ein Aufrufereignis aus Lambda abzurufen, übergibt das Ereignis an den Handler und sendet die Antwort zurück an Lambda. Um die Anforderungs-ID abzurufen, speichert die Laufzeit die Header aus der API-Antwort in einer temporären Datei und liest den `Lambda-Runtime-Aws-Request-Id`-Header aus der Datei. Sie liest auch den `Lambda-Runtime-Invocation-Id` Header und gibt ihn in der Antwort wieder.

> **Note**
>
> Laufzeit besitzen zusätzliche Verantwortlichkeiten, darunter Fehlerbehandlung und Bereitstellung von Kontextinformationen für den Handler. Details hierzu finden Sie unter Voraussetzungen.

3. Erstellen Sie ein Skript für die Funktion. Das folgende Beispielskript definiert eine Handler-Funktion, die Ereignisdaten annimmt, protokolliert sie in `stderr` und gibt sie zurück.

Example function.sh

```
function handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

Das `runtime-tutorial`-Verzeichnis sollte jetzt wie folgt aussehen:

```
runtime-tutorial
# bootstrap
# function.sh
```

4. Konvertieren Sie die Dateien zu ausführbaren Dateien und fügen Sie sie einem ZIP-Dateiarchiv hinzu. Dies ist das Bereitstellungspaket.

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. Erstellen Sie eine Funktion mit dem Namen `bash-runtime`. Geben Sie für `--role` den ARN Ihrer Lambda-Ausführungsrolle ein.

```
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime
 provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. Die Funktion aufrufen.

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI

unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. Überprüfen Sie die Antwort.

```
cat response.txt
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
Echoing request: '{"text":"Hello"}'
```

Erstellen einer Ebene

Um Laufzeitcode und Funktionscode voneinander zu trennen, erstellen Sie eine Ebene, die nur die Laufzeit enthält. Mit Ebenen können Sie die Abhängigkeiten Ihrer Funktion unabhängig entwickeln. Sie können darüber hinaus die Speichernutzung reduzieren, wenn Sie dieselbe Ebene für mehrere Funktionen verwenden. Weitere Informationen finden Sie unter Verwaltung der Abhängigkeiten von Lambda Event Function mit Ebenen.

1. Erstellen Sie eine ZIP-Datei, die die `bootstrap`-Datei enthält.

```
zip runtime.zip bootstrap
```

2. Erstellen Sie eine Ebene mit dem publish-layer-version-Befehl.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

Hierdurch wird die erste Version der Ebene erstellt.

Aktualisieren der Funktion

Um die Laufzeitebene in der Funktion zu verwenden, konfigurieren Sie die Funktion für die Verwendung der Ebene und entfernen den Laufzeitcode aus der Funktion.

1. Aktualisieren Sie die Konfiguration der Funktion, um die Ebene einzubeziehen.

   ```
   aws lambda update-function-configuration --function-name bash-runtime \
   --layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
   ```

   Dadurch wird die Laufzeit zur Funktion im `/opt`-Verzeichnis hinzugefügt. Um sicherzustellen, dass Lambda die Laufzeit in der Ebene verwendet, müssen Sie die `boostrap`-Datei aus dem Bereitstellungspaket der Funktion entfernen, wie in den nächsten beiden Schritten gezeigt.

2. Erstellen Sie eine ZIP-Datei, die den Funktionscode enthält.

   ```
   zip function-only.zip function.sh
   ```

3. Aktualisieren Sie den Funktionscode so, dass er nur das Handler-Skript enthält.

   ```
   aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
   function-only.zip
   ```

4. Rufen Sie die Funktion auf, um zu überprüfen, ob sie mit der Laufzeitebene funktioniert.

   ```
   aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
    response.txt --cli-binary-format raw-in-base64-out
   ```

   Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

   Das Ergebnis sollte ungefähr wie folgt aussehen:

   ```
   {
       "StatusCode": 200,
       "ExecutedVersion": "$LATEST"
   }
   ```

5. Überprüfen Sie die Antwort.

```
cat response.txt
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
Echoing request: '{"text":"Hello"}'
```

Aktualisieren der Laufzeit

1. Um Informationen zur Ausführungsumgebung zu protokollieren, aktualisieren Sie das Laufzeitskript, sodass es Umgebungsvariablen ausgibt.

Example bootstrap

```
#!/bin/sh

set -euo pipefail

# Configure runtime to output environment variables
echo "##  Environment variables:"
env

# Load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Extract invocation ID for cross-wiring protection
  INVOCATION_ID=$(grep -Fi Lambda-Runtime-Invocation-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)
```

```
  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response, echoing the invocation ID
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE" \
    ${INVOCATION_ID:+--header "Lambda-Runtime-Invocation-Id: $INVOCATION_ID"}
done
```

2. Erstellen Sie eine ZIP-Datei mit der neuen Version der `bootstrap`-Datei.

```
zip runtime.zip bootstrap
```

3. Erstellt eine neue Version der `bash-runtime`-Ebene.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. Konfigurieren Sie die Funktion für die Verwendung der neuen Version der Ebene.

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

Freigeben der Ebene

Um einen Layer mit einem anderen zu teilen AWS-Konto, fügen Sie der ressourcenbasierten Richtlinie des Layers eine kontoübergreifende Berechtigungserklärung hinzu. Führen Sie den Befehl add-layer-version-permission aus und geben Sie die Konto-ID als `principal` an. In jeder Anweisung können Sie einem einzelnen Konto, allen Konten oder einer Organisation in AWS Organizations eine Berechtigung erteilen.

Im folgenden Beispiel wird dem Konto 111122223333 Zugriff auf Version 2 der `bash-runtime`-Ebene gewährt.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
```

```
  --output text
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Berechtigungen gelten nur für eine Version mit einer Ebene. Wiederholen Sie den Vorgang bei jeder Erstellung einer neuen Ebenenversion.

Bereinigen

Löschen Sie alle Versionen der Ebene.

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

Da die Funktion einen Verweis auf Version 2 der Ebene enthält, ist diese nach wie vor in Lambda vorhanden. Die Funktion funktioniert weiter. Es können jedoch keine weiteren Funktionen für die Verwendung der gelöschten Version konfiguriert werden. Wenn Sie die Liste der Ebenen in der Funktion ändern, müssen Sie eine neue Version angeben oder die gelöschte Ebene auslassen.

Löschen Sie die Funktion mit dem Befehl delete-function.

```
aws lambda delete-function --function-name bash-runtime
```

Open-Source-Repositorys

AWS Lambda bietet eine Vielzahl von Open-Source-Tools, -Bibliotheken und -Komponenten, mit denen Sie Ihre serverlosen Anwendungen erstellen, anpassen und optimieren können. Zu diesen Ressourcen gehören Runtime-Interface-Clients, Ereignisbibliotheken, Container-Basis-Images, Entwicklungstools und Beispielprojekte, die von verwaltet werden AWS und auf GitHub dieser verfügbar sind. Mithilfe dieser Open-Source-Repositorys können Sie die Funktionen von Lambda erweitern, benutzerdefinierte Laufzeiten erstellen, Ereignisse aus verschiedenen AWS Diensten verarbeiten und tiefere Einblicke in die Leistung Ihrer Funktion gewinnen. Diese Seite bietet einen Überblick über die wichtigsten Open-Source-Projekte, die die Lambda-Entwicklung unterstützen.

Laufzeit-Schnittstellenclients

Laufzeit-Schnittstellenclients (RICs) von Lambda sind Open-Source-Bibliotheken, die die Laufzeit-API implementieren und die Interaktion zwischen Ihrem Funktionscode und dem Lambda-Service verwalten. Diese Clients übernehmen den Empfang von Aufrufereignissen, die Weitergabe von Kontextinformationen und die Meldung von Fehlern.

Die von den verwalteten Laufzeiten und Container-Basisimages von Lambda verwendeten Laufzeit-Schnittstellenclients werden als Open Source veröffentlicht. Wenn Sie benutzerdefinierte Laufzeiten erstellen oder bestehende erweitern, können Sie diese Open-Source-Bibliotheken verwenden, um Ihre Implementierung zu vereinfachen. Die folgenden GitHub Open-Source-Repositorys enthalten den Quellcode für die RICs von Lambda:

- Node.js Laufzeitschnittstelle-Client
- Python-Laufzeit-Schnittstellenclient
- Java-Laufzeit-Schnittstellenclient
- Ruby-Laufzeit-Schnittstellenclient
- .NET-Laufzeit-Schnittstellenclient
- Rust-Laufzeitschnittstelle-Client
- Go-Laufzeit-Schnittstellenclient
- Swift Runtime Interface-Client (experimentell)
- C++-Laufzeitschnittstellenclient (experimentell)
- Lambda-Basis-Images

Weitere Informationen zur Verwendung dieser Clients zum Erstellen benutzerdefinierter Laufzeiten finden Sie unter the section called “Erstellen einer benutzerdefinierten Laufzeit”.

Ereignisbibliotheken

Lambda-Ereignisbibliotheken bieten Typdefinitionen und Hilfsprogramme für die Verarbeitung von Ereignissen aus verschiedenen AWS Diensten. Diese Bibliotheken unterstützen Sie beim Parsen und Verarbeiten von Ereignisdaten auf typsichere Weise, wodurch die Arbeit mit Ereignissen aus Services wie Amazon S3, Amazon DynamoDB und Amazon API Gateway vereinfacht wird.

AWS Stellt für kompilierte Sprachen die folgenden Ereignisbibliotheken bereit:

- Java-Ereignisbibliothek

- .NET-Ereignisbibliotheken
- Go-Ereignisbibliothek
- Rust-Ereignisbibliothek

Für interpretierte Sprachen wie Node.js Python und Ruby können Ereignisse direkt als JSON-Objekte analysiert werden, ohne dass eine separate Bibliothek erforderlich ist. Entwickler, die Python verwenden Node.js , können jedoch Powertools for verwenden AWS Lambda, das integrierte Schemas für AWS Ereignisse bereitstellt, die Typhinweise, Datenvalidierung und Funktionen bieten, die denen von kompilierten Sprachbibliotheken ähneln.

- Powertools für TypeScript
- Powertools für Python

Container-Basisimages

AWS stellt Open-Source-Container-Basis-Images bereit, die Sie als Ausgangspunkt für die Erstellung von Container-Images für Ihre Lambda-Ereignisfunktionen verwenden können. Diese Basisimages enthalten den Laufzeit-Schnittstellenclient und andere Komponenten, die zum Ausführen Ihrer Funktionen in der Lambda-Ausführungsumgebung erforderlich sind.

Weitere Informationen zu den verfügbaren Basisimages und deren Verwendung finden Sie im Repository für AWS -Basisimages sowie unter the section called “Container-Images”.

Entwicklungstools

AWS bietet zusätzliche Open-Source-Entwicklungstools, mit denen Sie Ihre Lambda-Funktionen erstellen und optimieren können:

Powertools für AWS Lambda

Powertools for AWS Lambda vereinfacht die serverlose Entwicklung mit wichtigen Hilfsprogrammen zur Vermeidung doppelter Verarbeitung, Stapelverarbeitung für die Verarbeitung mehrerer Datensätze und Kafka-Benutzerbibliothek. Diese Funktionen helfen Ihnen, die Komplexität des Codes und den Betriebsaufwand zu minimieren.

Sie können auch die integrierte Ereignisschemavalidierung, die strukturierte Protokollierung und Ablaufverfolgung sowie die Integration von Parameterspeichern verwenden, die darauf ausgelegt

sind, die Erstellung produktionsreifer Lambda-Funktionen zu beschleunigen und gleichzeitig gut konzipierte Best Practices zu befolgen AWS .

GitHub Repositorien:

- Python
- TypeScript
- Java
- .NET

Java-Entwicklungstools

- Java Profiler (experimentell) – Ein Tool zur Profilerstellung für Java-Lambda-Funktionen.
- Java-Bibliotheken – Ein Repository, das eine umfassende Sammlung von Java-Bibliotheken und -Tools für die Lambda-Entwicklung enthält, darunter wichtige Projekte wie JUnit-Testdienstprogramme und Profilerstellungstools.
- Serverloser Java-Container — Eine Bibliothek, mit der Sie bestehende Java-Anwendungen auf Lambda mit minimalen Änderungen ausführen können.

.NET-Entwicklungstools

AWS Lambda Das.NET-Repository stellt .NET-Bibliotheken und Tools für die Lambda-Entwicklung bereit, einschließlich wichtiger Projekte wie AWS Lambda Tools für die .NET CLI und .NET Core-Server zum Hosten von.NET Core-Anwendungen.

Beispielprojekte

Erkunden Sie eine umfassende Sammlung von Beispielprojekten und -anwendungen für Lambda in den Serverless-Land-Repositorys. Diese Beispiele veranschaulichen verschiedene Anwendungsfälle, Integrationsmuster und bewährte Verfahren für Lambda, die Ihnen den Einstieg in Ihre Serverless-Anwendungen erleichtern.

Konfigurieren AWS Lambda Event-Funktionen

Erfahren Sie, wie Sie die Kernfunktionen und Optionen für Ihre Lambda-Ereignisfunktion mithilfe der Lambda-API oder -Konsole konfigurieren.

.zip-Dateiarchive

Erstellen Sie ein Paket zur Bereitstellung von Lambda-Funktionen, wenn Sie Abhängigkeiten, benutzerdefinierte Laufzeitebenen oder Dateien außerhalb Ihres Funktionscodes einbeziehen möchten. Das Bereitstellungspaket ist ein ZIP-Dateiarchiv, das Ihren Funktionscode und Ihre Abhängigkeiten enthält.

Container-Images

Verwenden Sie Container-Images, um Ihren Funktionscode und Ihre Abhängigkeiten zu verpacken, wenn Sie mehr Kontrolle über den Erstellungsprozess benötigen oder wenn Ihre Funktion benutzerdefinierte Laufzeitkonfigurationen erfordert. Mit Tools wie Docker CLI können Sie Lambda-Funktionen als Container-Images erstellen, testen und bereitstellen.

Self-managed S3-Codespeicher

Konfigurieren Sie Ihre Funktionen und Ebenen so, dass Sie Ihren eigenen S3-Bucket für die Codespeicherung verwenden. Lambda referenziert Ihren Code direkt aus Ihrem S3-Bucket, wodurch Lambda-managed Speicherbeschränkungen vermieden und die Zeit bis zum ersten Aufruf reduziert wird.

Arbeitsspeicher

Hier erfahren Sie, wie und wann das Funktionsgedächtnis erweitert wird.

Flüchtiger Speicher

Erfahren Sie, wie und wann Sie die temporäre Speicherkapazität Ihrer Funktion erhöhen können.

Timeout (Zeitüberschreitung)

Erfahren Sie, wie und wann Sie den Timeout-Wert Ihrer Funktion erhöhen können.

Umgebungsvariablen

Sie können Ihren Funktionscode portabel machen und Geheimnisse aus Ihrem Code heraushalten, indem Sie sie mithilfe von Umgebungsvariablen in der Konfiguration Ihrer Funktion speichern.

Ausgehende Netzwerke

Sie können Ihre Lambda-Funktion mit AWS Ressourcen in einer Amazon VPC verwenden. Wenn Sie Ihre Funktion mit einer VPC verbinden, erhalten Sie Zugriff auf Ressourcen in einem privaten Subnetz wie relationale Datenbanken und Caches.

Eingehende Netzwerke

Sie können einen Schnittstellen-VPC-Endpunkt verwenden, um Ihre Lambda-Funktionen aufzurufen, ohne das öffentliche Internet zu nutzen.

Dateisystem

Sie können Ihre Lambda-Funktion verwenden, um ein Amazon EFS in ein lokales Verzeichnis einzubinden. Ein Dateisystem ermöglicht es Ihrem Funktionscode, sicher und mit hoher Gleichzeitigkeit auf gemeinsam genutzte Ressourcen zuzugreifen und diese zu ändern.

Aliasnamen

Sie können Ihre Clients so konfigurieren, dass sie eine bestimmte Lambda-Funktionsversion aufrufen, indem Sie einen Alias verwenden, anstatt den Client zu aktualisieren.

Versionen

Durch die Veröffentlichung einer Version Ihrer Funktion können Sie Ihren Code und Ihre Konfiguration als separate Ressource speichern, die nicht geändert werden kann.

Tags

Verwenden Sie Tags, um die attributebasierte Zugriffskontrolle (ABAC) zu aktivieren, Ihre Lambda-Funktionen zu organisieren und Berichte über Ihre Funktionen mithilfe der Dienste AWS Cost Explorer oder AWS Billing and Cost Management zu filtern und zu generieren.

Antwort-Streaming

Sie können Ihre Lambda-Funktions-URLs so konfigurieren, dass sie Antwort-Nutzlasten zurück an Clients streamen. Antwort-Streaming kann für latenzempfindliche Anwendungen von Vorteil sein, da es die Leistung in der Zeit bis zum ersten Byte (TTFB) verbessert. Dies liegt daran, dass Sie Teilantworten an den Client zurücksenden können, sobald sie verfügbar sind. Darüber hinaus können Sie Response-Streaming verwenden, um Funktionen zu erstellen, die größere Nutzlasten zurückgeben.

Metadaten-Endpunkt

Verwenden Sie den Lambda-Metadaten-Endpunkt, um zu ermitteln, in welcher Availability Zone Ihre Funktion ausgeführt wird. So können Sie die Latenz optimieren, indem Sie zu Ressourcen derselben AZ weiterleiten und Resilienzmuster implementieren AZ-aware .

Bereitstellung von Lambda-Ereignisfunktionen als.zip-Dateiarchive

Wenn Sie eine Lambda-Ereignisfunktion erstellen, packen Sie Ihren Funktionscode in ein Bereitstellungspaket. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und .zip-Archive. Der Workflow zum Erstellen einer Funktion hängt vom Typ des Bereitstellungspakets ab. Zur Konfiguration einer Funktion, die als Container-Image definiert ist, siehe the section called “Container-Images”.

Sie können die Lambda-Konsole und die Lambda-API verwenden, um eine mit einem ZIP-Dateiarchiv definierte Funktion zu erstellen. Sie können auch eine aktualisierte ZIP-Datei uploaden, um den Funktionscode zu ändern.

Wenn Sie eine ZIP-Datei hochladen, speichert Lambda standardmäßig eine Kopie Ihres Quellcodes in Lambda-managed Speicher. Mit selbstverwalteter S3-Code-Speicher, Lambda liest Ihren Quellcode während der Erstellung und Aktualisierung von Funktionen aus Ihrem S3-Bucket und speichert keine separate Kopie Ihres Quellcodes. Lambda erstellt immer noch eine interne optimierte Darstellung Ihres Codes für den Aufruf. Auf Ihren Quellcode in S3 wird nicht bei jedem Kaltstart zugegriffen.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

Themen

- Erstellen der Funktion
- Verwenden des Konsolencode-Editors
- Aktualisieren des Funktionscodes
- Ändern der Laufzeit
- Ändern der Architektur
- Verwenden der Lambda-API
- Herunterladen des Funktionscodes
- CloudFormation

- Verschlüsselung von Lambda-.zip-Bereitstellungspaketen

Erstellen der Funktion

Wenn Sie eine Funktion erstellen, die mit einem ZIP-Dateiarchiv definiert ist, wählen Sie eine Codevorlage, die Sprachversion und die Ausführungsrolle für die Funktion aus. Sie fügen Ihren Funktionscode hinzu, nachdem Lambda die Funktion erstellt hat.

So erstellen Sie die Funktion

1. Öffnen Sie das Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie Event-Funktion erstellen.
3. Wählen Sie Von Grund auf neu erstellen oder Vorlage verwenden, um Ihre Funktion zu erstellen.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Funktionsname den Funktionsnamen ein. Funktionsnamen sind auf eine Länge von 64 Zeichen beschränkt.
   b. Wählen Sie für Laufzeit die Sprachversion aus, die für Ihre Funktion verwendet werden soll.
   c. (Optional) Für-Architekturwählen Sie die Befehlssatz-Architektur aus, die für Ihre Funktion verwendet werden soll. Die Standardarchitektur ist x86_64. Stellen Sie beim Erstellen des Bereitstellungspakets für Ihre Funktion sicher, dass es damit kompatibel istBefehlssatz-Architekturaus.
5. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
6. (Optional) Erweitern Sie den Abschnitt Advanced Settings (Erweiterte Einstellungen). Sie können eine Codesignatur-Konfiguration für die Funktion wählen. Sie können auch eine (Amazon VPC) für die Funktion konfigurieren, auf die zugegriffen werden soll.
7. Wählen Event-Funktion erstellen.

Lambda erstellt die neue Funktion. Sie können nun die Konsole verwenden, um den Funktionscode hinzuzufügen und andere Funktionsparameter und -funktionen zu konfigurieren. Anweisungen zur Codebereitstellung finden Sie auf der Handler-Seite für die Laufzeit, die Ihre Funktion verwendet.

Node.js

Stellen Sie Node.js Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit

Python

- Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions

Ruby

- Stellen Sie Ruby Lambda Event Functions mit.zip-Dateiarchiven bereit

Java

- Stellen Sie Java Lambda Event Functions mit.zip- oder JAR-Dateiarchiven bereit

Go

- Stellen Sie Go Lambda Event Functions mit.zip-Dateiarchiven bereit

C#

- C#-Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven erstellen und bereitstellen

PowerShell

- Stellen Sie PowerShell Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit

Verwenden des Konsolencode-Editors

Die Konsole erstellt eine Lambda-Funktion mit einer einzigen Quelldatei. Bei Skriptsprachen können Sie diese Datei bearbeiten und mit dem integrierten Code-Editor weitere Dateien hinzufügen. Klicken Sie auf Save (Speichern), um die Änderungen zu speichern. Um Ihren Code auszuführen, wählen Sie Test.

Wenn Sie Ihren Funktionscode speichern, erstellt die Lambda-Konsole ein Bereitstellungspaket für das ZIP-Dateiarchiv. Wenn Sie Ihren Funktionscode außerhalb der Konsole (mit einer IDE) entwickeln, müssen Sie ein Bereitstellungspaket erstellen, um Ihren Code in die Lambda-Funktion hochzuladen.

Aktualisieren des Funktionscodes

Für Skriptsprachen (Node.js, Python und Ruby), können Sie Ihren Funktionscode im eingebetteten Code-Editor bearbeiten. Wenn der Code größer als 3 MB ist, oder wenn Sie Bibliotheken hinzufügen müssen, oder für Sprachen, die der Editor nicht unterstützt (Java, Go, C#), müssen Sie Ihren Funktionscode als .zip-Archiv uploaden. Wenn das ZIP-Dateiarchiv kleiner als 50 MB ist, können Sie

das ZIP-Dateiarchiv von Ihrem lokalen Computer uploaden. Wenn die Datei größer als 50 MB ist, müssen Sie die Datei aus einem Amazon S3-Bucket in die Funktion uploaden.

So können Sie den Funktionscode als .zip-Archiv uploaden

1. Öffnen Sie den Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Wählen Sie unter Codequelle die Option Upload von aus.
4. Wählen Sie .zip-Datei und dann Hochladen.
   - Wählen Sie in der Dateiauswahl die neue Image-Version aus, wählen Sie Öffnen und dann Speichern.
5. (Alternative zu Schritt 4) Wählen Sie den Amazon-S3-Standort.
   a. Geben Sie in das Textfeld die S3-Link-URL des ZIP-Dateiarchivs ein.
   b. Unter Code-Speichermodus, wählen Sie eine der folgenden Optionen:
      - Kopiermodus (Standard)
      - Referenzmodus
   c. Wählen Sie Speichern.

Bei der Aktualisierung des Funktionscodes aus einem Amazon S3-Bucket mit dem AWS CLI, können Sie den Codespeichermodus angeben. Stellen Sie ein `S3ObjectStorageMode` zu `REFERENCE` zu verwenden selbstverwalteter S3-Code-Speicher, oder `COPY` (Standard) zu verwenden Lambda-managed Speicher. Sie können den Speichermodus bei jedem Update ändern.

```
aws lambda update-function-code \
  --function-name my-function \
  --s3-bucket my-bucket \
  --s3-key my-function.zip \
  --s3-object-version abc123 \
  --s3-object-storage-mode REFERENCE
```

Ändern der Laufzeit

Wenn Sie die Funktionskonfiguration aktualisieren, um eine neue Laufzeit zu verwenden, müssen Sie möglicherweise den Funktionscode aktualisieren, damit er mit der neuen Laufzeit kompatibel ist. Wenn Sie die Funktionskonfiguration aktualisieren, um eine andere Laufzeit zu

verwenden,mussstellen Sie neuen Funktionscode bereit, der mit Laufzeit und Architektur kompatibel ist. Anweisungen zum Erstellen eines Bereitstellungspakets für den Funktionscode finden Sie auf der Handler-Seite für die Laufzeit, die die Funktion verwendet.

Die Node.js Die Basis-Images 20, Python 3.12, Java 21, .NET 8, Ruby 3.3 und höher basieren auf dem minimalen Container-Image von Amazon Linux 2023. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie `glibc`. Weitere Informationen finden Sie unter Introducing the Amazon Linux 2023 runtime for AWS Lambda im AWS -Compute-Blog.

So ändern Sie die Laufzeit

1. Öffnen Sie das Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Blättern Sie nach unten bis zum Abschnitt Laufzeiteinstellungen, der sich unter dem Code-Editor befindet.
4. Wählen Sie Bearbeiten aus.
    a. Wählen Sie für Runtime (Laufzeit) die Laufzeit-ID aus.
    b. Geben Sie im Feld Handler den Dateinamen und en Handler für die Funktion ein.
    c. Für-Architekturwählen Sie die Befehlssatz-Architektur aus, die für Ihre Funktion verwendet werden soll.
5. Wählen Sie Speichern.

Ändern der Architektur

Bevor Sie die Befehlssatzarchitektur ändern können, müssen Sie sicherstellen, dass der Code Ihrer Funktion mit der Zielarchitektur kompatibel ist.

Wenn du verwendest Node.js, Python oder Ruby und Sie bearbeiten Ihren Funktionscode im eingebetteten Editor, wird der vorhandene Code möglicherweise ohne Änderung ausgeführt.

Wenn Sie jedoch Ihren Funktionscode mithilfe eines Bereitstellungspakets im ZIP-Dateiarchiv angeben, müssen Sie ein neues ZIP-Dateiarchiv vorbereiten, das korrekt für die Ziel-Laufzeit- und Anleitungssatz-Architektur kompiliert und erstellt wird. Anweisungen dazu finden Sie auf der Handler-Seite für Ihre Funktions-Laufzeit.

So ändern Sie die Befehlssatz-Architektur

1. Öffnen Sie den Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Wählen Sie unter Laufzeiteinstellungen die Option Bearbeiten.
4. Für-Architekturwählen Sie die Befehlssatz-Architektur aus, die für Ihre Funktion verwendet werden soll.
5. Wählen Sie Speichern.

Verwenden der Lambda-API

Um eine Funktion zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwendet, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

Herunterladen des Funktionscodes

Sie können das aktuelle unveröffentlichte herunterladen (`$LATEST`) Version Ihres Funktionscodes (.zip) von der Lambda-Konsole. Um dies zu tun, müssen Sie zunächst über die folgenden IAM-Berechtigungen verfügen:

- `iam:GetPolicy`
- `iam:GetPolicyVersion`
- `iam:GetRole`
- `iam:GetRolePolicy`
- `iam:ListAttachedRolePolicies`
- `iam:ListRolePolicies`
- `iam:ListRoles`

So laden Sie den Funktionscode (.zip) herunter

1. Öffnen Sie das Seite „Event-Funktionen“ der Lambda-Konsole.

2. Wählen Sie die Funktion aus, für die Sie den Funktionscode (.zip) herunterladen möchten.
3. Wählen Sie in der Funktionsübersicht die Schaltfläche Herunterladen und anschließend Funktionscode herunterladen (.zip).

    - Wählen Sie alternativ Laden Sie herunter AWS SAM Ordner um eine SAM-Vorlage zu generieren und herunterzuladen, die auf der Konfiguration Ihrer Funktion basiert. Sie können auch Beide herunterladen wählen, um sowohl die ZIP-Vorlage als auch die SAM-Vorlage herunterzuladen.

CloudFormation

Sie können verwenden CloudFormation um eine Lambda-Funktion zu erstellen, die ein ZIP-Dateiarchiv verwendet. In deinem CloudFormation Vorlage, die `AWS::Lambda::Function` Ressource spezifiziert die Lambda-Funktion. Für Beschreibungen der Eigenschaften finden Sie in der `AWS::Lambda::Function` Ressource, siehe AWS::Lambda::Funktion in der AWS CloudFormation Benutzerleitfaden.

Legen Sie in der `AWS::Lambda::Function`-Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als Zip-Datei-Archiv definiert ist:

- AWS::Lambda::Funktion
    - PackageType — Auf gesetzt `Zip`.
    - Code – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Textfelder `S3Bucket` und `S3Key`. Für Node.js oder Python, Sie können den Inline-Quellcode Ihrer Lambda-Funktion bereitstellen.
    - Laufzeit – Legt den Laufzeitwert fest.
    - Architektur — Setzen Sie den Architekturwert auf `arm64` zu verwenden AWS Graviton-Prozessoren. Der Standardwert ist`x86_64`aus.
    - S3ObjectStorageMode — (Optional) Auf eingestellt `REFERENCE` zu verwenden selbstverwalteter S3-Code-Speicher. Eingestellt auf `COPY` (Standard) zu verwenden Lambda-managed Speicher.

Verschlüsselung von Lambda-.zip-Bereitstellungspaketen

AWS Lambda bietet immer serverseitige Verschlüsselung im Ruhezustand für .zip-Bereitstellungspakete und Details zur Funktionskonfiguration mit einem. AWS KMS key Standardmäßig verwendet Lambda einen AWS-eigener Schlüssel. Wenn dieses Standardverhalten

zu Ihrem Arbeitsablauf passt, müssen Sie nichts weiter einrichten. AWS berechnet Ihnen keine Gebühren für die Verwendung dieses Schlüssels.

Wenn Sie möchten, können Sie stattdessen einen vom AWS KMS Kunden verwalteten Schlüssel bereitstellen. Sie können dies tun, um die Drehung des KMS-Schlüssels zu steuern oder die Anforderungen Ihrer Organisation für die Verwaltung von KMS-Schlüsseln zu erfüllen. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, können nur Benutzer Ihres Kontos mit Zugriff auf den KMS-Schlüssel den Code oder die Konfiguration der Funktion anzeigen oder verwalten.

Für vom Kunden verwaltete Schlüssel fallen AWS KMS Standardgebühren an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Erstellen eines kundenseitig verwalteten Schlüssels

Sie können einen symmetrischen, vom Kunden verwalteten Schlüssel mithilfe der APIs AWS-Managementkonsole oder erstellen. AWS KMS

Einen symmetrischen kundenverwalteten Schlüssel erstellen

Befolgen Sie die Schritte zur Erstellung symmetrischer Verschlüsselung Erstellung symmetrischer KMS-Schlüssel mit symmetrischer Verschlüsselung im AWS Key Management Service - Entwicklerhandbuch.

Berechtigungen

Schlüsselrichtlinie

Schlüsselrichtlinien steuern den Zugriff auf den vom Kunden verwalteten Schlüssel. Jeder vom Kunden verwaltete Schlüssel muss über genau eine Schlüsselrichtlinie verfügen, die aussagt, wer den Schlüssel wie verwenden kann. Weitere Informationen finden Sie unter So ändern Sie eine Schlüsselrichtlinie im AWS Key Management Service -Entwicklerhandbuch.

Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, um ein .zip-Bereitstellungspaket zu verschlüsseln, fügt Lambda dem Schlüssel keinen Zuschuss hinzu. Stattdessen muss Ihre AWS KMS Schlüsselrichtlinie es Lambda ermöglichen, die folgenden AWS KMS API-Operationen in Ihrem Namen aufzurufen:

- km: GenerateDataKey
- kms:Decrypt

Die folgende Beispielschlüsselrichtlinie ermöglicht es allen Lambda-Funktionen im Konto 111122223333, die erforderlichen AWS KMS Operationen für den angegebenen vom Kunden verwalteten Schlüssel aufzurufen:

Example AWS KMS wichtige Richtlinie

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-1:111122223333:key/key-id",
            "Condition": {
                "StringLike": {
                "kms:EncryptionContext:aws:lambda:FunctionArn":
 "arn:aws:lambda:us-east-1:111122223333:function:*"
                }
            }
        }
    ]
}
```

Weitere Informationen zur Fehlerbehebung beim Schlüsselzugriff finden Sie im AWS Key Management Service -Entwicklerhandbuch.

Prinzipalberechtigungen

Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, um ein .zip-Bereitstellungspaket zu verschlüsseln, können nur Prinzipale mit Zugriff auf diesen Schlüssel auf das .zip-Bereitstellungspaket zugreifen. Principals, die keinen Zugriff auf den vom Kunden verwalteten Schlüssel haben, können beispielsweise das .zip-Paket nicht über die vorsignierte S3-URL

herunterladen, die in der Antwort enthalten ist. GetFunction Im Abschnitt `Code` der Antwort wird ein `AccessDeniedException` zurückgegeben.

Example AWS KMS AccessDeniedException

```
{
    "Code": {
        "RepositoryType": "S3",
        "Error": {
            "ErrorCode": "AccessDeniedException",
            "Message": "KMS access is denied. Check your KMS permissions. KMS
 Exception: AccessDeniedException KMS Message: User: arn:aws:sts::111122223333:assumed-
role/LambdaTestRole/session is not authorized to perform: kms:Decrypt on resource:
 arn:aws:kms:us-east-1:111122223333:key/key-id with an explicit deny in a resource-
based policy"
        },
        "SourceKMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id"
    },
 ...
```

Weitere Informationen zu Berechtigungen für AWS KMS Schlüssel finden Sie unter Authentifizierung und Zugriffskontrolle für. AWS KMS

Verwenden Sie einen vom Kunden verwalteten Schlüssel für Ihr .zip-Bereitstellungspaket

Verwenden Sie die folgenden API-Parameter, um vom Kunden verwaltete Schlüssel für .zip-Bereitstellungspakete zu konfigurieren:

- SourceKMSKeyArn: Verschlüsselt das Quell-.zip-Bereitstellungspaket (die Datei, die Sie hochladen).
- KMSKeyArn: Verschlüsselt Umgebungsvariablen und Lambda-Snapshots SnapStart.
- DurableConfig.KMSKeyArn: Verschlüsselt dauerhafte Ausführungsdaten. Dieser Schlüssel ist unabhängig von der `KMSKeyArn` Funktionsebene.

Wenn `SourceKMSKeyArn` und `KMSKeyArn` beide angegeben sind, verwendet Lambda den Schlüssel `KMSKeyArn`, um die entpackte Version des Pakets zu verschlüsseln, das Lambda zum Aufrufen der Funktion verwendet. Wenn `SourceKMSKeyArn` angegeben ist, aber `KMSKeyArn` nicht, verwendet Lambda ein Von AWS verwalteter Schlüssel, um die entpackte Version des Pakets zu verschlüsseln.

> **Note**
>
> Für Funktionen und Ebenen, die selbstverwalteten S3-Codespeicher verwenden, speichert Lambda keine Kopie Ihres Quellcodes. Ihr Code verbleibt in Ihrem S3-Bucket, und die Verschlüsselung Ihres Quellcodes im Ruhezustand wird von Ihrer S3-Bucket-Konfiguration verwaltet. Lambda wird verwendet`KMSKeyArn`, um die entpackte Version des Pakets zu verschlüsseln, das Lambda zum Aufrufen der Funktion verwendet. Weitere Informationen finden Sie unter Schützen von Daten mit serverseitiger Verschlüsselung im Amazon Simple Storage Service-Benutzerhandbuch.

Lambda console

So fügen Sie beim Erstellen einer Funktion die kundenverwaltete Schlüsselverschlüsselung hinzu

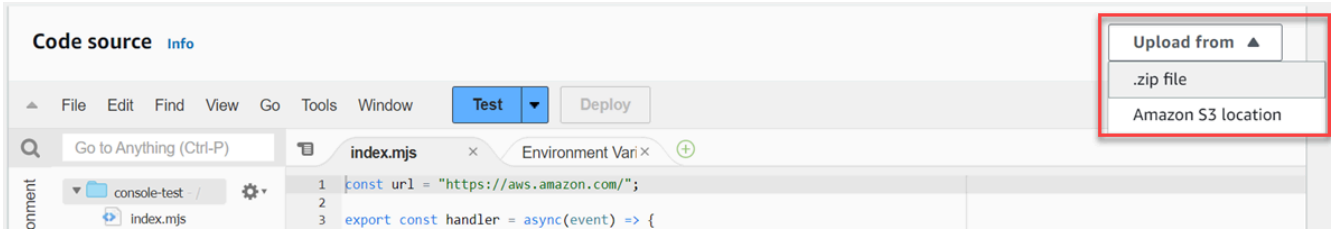
1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Klicken Sie auf Ohne Vorgabe erstellen oder Container-Image.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Funktionsname den Funktionsnamen ein.
   b. Wählen Sie für Laufzeit die Sprachversion aus, die für Ihre Funktion verwendet werden soll.
5. Erweitern Sie Erweiterte Einstellungen und wählen Sie dann Verschlüsselung mit einem vom AWS KMS Kunden verwalteten Schlüssel aktivieren aus.
6. Wählen Sie einen kundenverwalteten Schlüssel.
7. Wählen Sie „Event-Funktion erstellen“.

Um die vom Kunden verwaltete Schlüsselverschlüsselung zu entfernen oder einen anderen Schlüssel zu verwenden, müssen Sie das .zip-Bereitstellungspaket erneut hochladen.

So fügen Sie einer bestehenden Funktion eine vom Kunden verwaltete Schlüsselverschlüsselung hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.

3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie eine .zip-Datei oder einen Amazon-S3-Speicherort.



5. Laden Sie die Datei hoch oder geben Sie den Amazon-S3-Speicherort ein.
6. Wählen Sie Verschlüsselung mit einem vom AWS KMS Kunden verwalteten Schlüssel aktivieren aus.
7. Wählen Sie einen kundenverwalteten Schlüssel.
8. Wählen Sie Speichern.

AWS CLI

So fügen Sie beim Erstellen einer Funktion die kundenverwaltete Schlüsselverschlüsselung hinzu

Im folgenden Beispiel wird create-function erstellt:

- `--code`: Gibt den lokalen Pfad zum ZIP-Bereitstellungspaket (`ZipFile`) und den vom Kunden verwalteten Schlüssel zur Verschlüsselung an (`SourceKMSKeyArn`).
- `--kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zur Verschlüsselung der Umgebungsvariablen und der entpackten Version des Bereitstellungspakets an.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code ZipFile=fileb://myFunction.zip,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

Im folgenden Beispiel wird create-function erstellt:

- `--code`: Gibt den Speicherort der .zip-Datei in einem Amazon S3 S3-Bucket (`S3Bucket`,`S3Key`,`S3ObjectVersion`) und den vom Kunden verwalteten Schlüssel zur Verschlüsselung an (`SourceKMSKeyArn`).
- `--kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zur Verschlüsselung der Umgebungsvariablen und der entpackten Version des Bereitstellungspakets an.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

So fügen Sie einer bestehenden Funktion eine vom Kunden verwaltete Schlüsselverschlüsselung hinzu

Im folgenden Beispiel des update-function-code:

- `--zip-file`: Gibt den lokalen Pfad zum .zip-Bereitstellungspaket an.
- `--source-kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zum Verschlüsseln der komprimierten Version des Bereitstellungspakets an. Lambda verwendet einen AWS eigenen Schlüssel, um das entpackte Paket für Funktionsaufrufe zu verschlüsseln. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden möchten, um die entpackte Version des Pakets zu verschlüsseln, führen Sie den Befehl update-function-configuration mit der `--kms-key-arn`-Option aus.

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip \
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

Im folgenden Beispiel des update-function-code:

- `--s3-bucket`: Gibt den Speicherort der .zip-Datei in einem Amazon-S3-Bucket an.
- `--s3-key`: Gibt den Amazon-S3-Schlüssel des Bereitstellungspakets an.

- `--s3-object-version`: Für versionierte Objekte, die Version des zu verwendenden Bereitstellungspaketobjekts.
- `--source-kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zum Verschlüsseln der komprimierten Version des Bereitstellungspakets an. Lambda verwendet einen AWS eigenen Schlüssel, um das entpackte Paket für Funktionsaufrufe zu verschlüsseln. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden möchten, um die entpackte Version des Pakets zu verschlüsseln, führen Sie den Befehl update-function-configuration mit der `--kms-key-arn`-Option aus.

```
aws lambda update-function-code \
  --function-name myFunction \
  --s3-bucket amzn-s3-demo-bucket \
  --s3-key myFileName.zip \
  --s3-object-version myObject Version
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

So entfernen Sie die vom Kunden verwaltete Schlüsselverschlüsselung aus einer bestehenden Funktion

Im folgenden Beispiel für update-function-code gibt `--zip-file` den lokalen Pfad zu dem .zip-Bereitstellungspaket an. Wenn Sie diesen Befehl ohne die `--source-kms-key-arn` Option ausführen, verwendet Lambda einen AWS eigenen Schlüssel, um die komprimierte Version des Bereitstellungspakets zu verschlüsseln.

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip
```

Erstellen Sie eine Lambda-Ereignisfunktion mithilfe eines Container-Images

Der Code Ihrer AWS Lambda Funktion besteht aus Skripten oder kompilierten Programmen und deren Abhängigkeiten. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und ZIP-Dateiarchiven.

Es gibt drei Möglichkeiten, ein Container-Image für eine Lambda-Funktion zu erstellen:

- Mit einem AWS Basisbild für Lambda

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Mit einem AWS OS-only Basisbild

  AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.

- Mit einem nicht-AWS Basisbild

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Um eine Lambda-Funktion aus einem Container-Image zu erstellen, erstellen Sie Ihr Image lokal und laden es in ein Amazon Elastic Container Registry (Amazon ECR)-Repository hoch. Wenn Sie ein von einem AWS Marketplace-Verkäufer bereitgestelltes Container-Image verwenden, müssen Sie das Image zuerst in Ihr privates Amazon-ECR-Repository klonen. Geben Sie dann den Repository-URI an, wenn Sie die Funktion erstellen. Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden. Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

> **Note**
>
> Lambda unterstützt Amazon-ECR-FIPS-Endpunkte für Container-Images nicht. Wenn Ihr Repository-URI ein `ecr-fips` enthält, verwenden Sie einen FIPS-Endpunkt. Beispiel: `111122223333.dkr.ecr-fips.us-east-1.amazonaws.com`.

Auf dieser Seite werden die Basis-Image-Typen und die Anforderungen für die Erstellung von Lambda-compatible Container-Images erläutert.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

Themen

- Voraussetzungen
- Mit einem AWS Basisbild für Lambda
- Mit einem AWS OS-only Basisbild
- Mit einem nicht-AWS Basisbild
- Laufzeitschnittstellen-Clients
- Amazon-ECR-Berechtigungen
- Lebenszyklus der Funktion

Voraussetzungen

Installieren Sie die AWS CLI -Version 2 und die Docker-CLI. Beachten Sie außerdem die folgenden Anforderungen:

- Das Container-Image muss die Verwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten implementieren. Die AWS Open-Source-Laufzeitschnittstellen-Clients implementieren die API. Sie können Ihrem bevorzugten Basis-Image einen Laufzeitschnittstellen-Client hinzufügen, damit es mit Lambda kompatibel ist.
- Das Container-Image muss auf einem schreibgeschützten Dateisystem laufen können. Ihr Funktionscode kann auf ein beschreibbares `/tmp`-Verzeichnis mit einem Speicherplatz zwischen 512 MB bis 10 240 MB, in 1-MB-Schritten, zugreifen.
- Der Lambda-Standardbenutzer muss in der Lage sein, alle Dateien zu lesen, die zum Ausführen Ihres Funktionscodes erforderlich sind. Lambda folgt den bewährten Methoden für die Sicherheit, indem es einen Standard-Linux-Benutzer mit den geringsten Berechtigungen definiert. Das bedeutet, dass Sie in Ihrem Dockerfile keinen BENUTZER angeben müssen. Stellen Sie sicher, dass Ihr Anwendungscode nicht auf Dateien angewiesen ist, von denen andere Linux-Benutzer nicht ausgeführt werden können.
- Lambda unterstützt nur Linux-based Container-Images.
- Lambda bietet Multi-Architektur-Basis-Images. Das Image, das Sie für Ihre Funktion erstellen, muss jedoch nur auf eine der Architekturen abzielen. Lambda unterstützt keine Funktionen, die Container-Images mit mehreren Architekturen verwenden.

Mit einem AWS Basisbild für Lambda

Sie können eines der AWS -Basis-Images für Lambda verwenden, um das Container-Image für Ihren Funktionscode zu erstellen. Die Basis-Images sind mit einer Sprachlaufzeit und anderen Komponenten vorgeladen, die zum Ausführen eines Container-Images in Lambda erforderlich sind. Sie fügen Ihren Funktionscode und Ihre Abhängigkeiten dem Basis-Image hinzu und verpacken es dann als Container-Image.

AWS stellt regelmäßig Updates für die AWS Basis-Images für Lambda bereit. Wenn Ihr Dockerfile den Image-Namen in der FROM-Eigenschaft enthält, ruft Ihr Docker-Client die aktuelle Version des Images aus dem Amazon-ECR-Repository ab.. Um das aktualisierte Basis-Image verwenden zu können, müssen Sie Ihr Container-Image neu erstellen und den Funktionscode aktualisieren.

Die Basis-Images Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3 und höher basieren auf dem minimalen Container-Image von Amazon Linux 2023. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie `glibc`.

AL2023-based Bilder verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, was der Standard-Paketmanager in Amazon Linux 2 ist. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2023-based Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2023 und Amazon Linux 2 finden Sie unter Introducing the Amazon Linux 2023 runtime for AWS Lambda im AWS -Compute-Blog.

> **Note**
>
> Um AL2023-based Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Um ein Container-Image mit einem AWS Basis-Image zu erstellen, wählen Sie die Anweisungen für Ihre bevorzugte Sprache aus:

- Node.js
- TypeScript(verwendet ein Node.js Basis-Image)
- Python
- Java
- Go
- .NET
- Ruby

Mit einem AWS OS-only Basisbild

AWS OS-only Basis-Images enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust und für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie z. B. Node.js 19. Sie können auch OS-only Basis-Images verwenden, um eine

benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
al2023	OS-only Laufzeit	Amazon Linux 2023	Dockerfile für Runtime auf OS-only GitHub	30. Juni 2029

Öffentliche Galerie von Amazon Elastic Container Registry: gallery.ecr. aws/lambda/bereitgestellt

Mit einem nicht-AWS Basisbild

Lambda unterstützt jedes Image, das einem der folgenden Image-Manifestformate entspricht:

- Docker Image Manifest V2 Schema 2 (mit Docker-Version 1.10 und neuer)
- Open Container Initiative (OCI)-Spezifikationen (v1.0.0 und höher)

Lambda unterstützt eine maximale unkomprimierte Image-Größe von 10 GB, einschließlich aller Ebenen.

> **Note**
>
> - Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.
> - Für eine optimale Leistung sollte die Größe Ihres Bildmanifests unter 25 400 Byte liegen. Um die Größe des Bildmanifests zu reduzieren, minimieren Sie die Anzahl der Ebenen in Ihrem Bild und reduzieren Sie Anmerkungen.

Laufzeitschnittstellen-Clients

Wenn Sie ein OS-only Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie einen Runtime-Interface-Client in Ihr Image aufnehmen. Der Runtime-Schnittstellenclient muss den erweiternVerwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten, der die Interaktion

zwischen Lambda und Ihrem Funktionscode verwaltet. AWS stellt Open-Source-Runtime-Interface-Clients für die folgenden Sprachen bereit:

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rust —

Wenn Sie eine Sprache verwenden, für die kein Runtime-Interface-Client AWS zur Verfügung steht, müssen Sie Ihren eigenen erstellen.

Amazon-ECR-Berechtigungen

Bevor Sie eine Lambda-Funktion aus einem Container-Image erstellen, müssen Sie das Image lokal erstellen und es in ein Amazon-ECR-Repository hochladen. Geben Sie beim Erstellen der Funktion den URI des Amazon-ECR-Repositorys an.

Stellen Sie sicher, dass die Berechtigungen für den Benutzer oder die Rolle, die die Funktion erstellt`GetRepositoryPolicy`, `SetRepositoryPolicy``BatchGetImage`, und `GetDownloadUrlForLayer` beinhalten.

Verwenden Sie beispielsweise die IAM-Konsole, um eine Rolle mit der folgenden Richtlinie zu erstellen:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
```

```
        "ecr:GetRepositoryPolicy",
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
}
```

Die Berechtigungen, die Lambda zum Abrufen Ihres Container-Images benötigt, hängen davon ab, ob sich das Amazon ECR-Repository im selben AWS Konto wie die Funktion oder in einem anderen Konto befindet. In den folgenden Abschnitten werden die Anforderungen für jedes Szenario erläutert.

> **Note**
>
> In IAM muss für den Zugriff auf dasselbe Konto nur eine Seite die Erlaubnis erteilen — entweder die identitätsbasierte Richtlinie (für die Rolle) oder die ressourcenbasierte Richtlinie (für das Amazon ECR-Repository). Cross-account Für den Zugriff müssen beide Seiten eine Genehmigung erteilen — die identitätsbasierte Richtlinie für die Rolle im verbrauchenden Konto UND die ressourcenbasierte Richtlinie für das Amazon ECR-Repository im Eigentümerkonto müssen beide die Aktion zulassen.

Same-account Amazon ECR-Repository-Richtlinien

Für eine Funktion in demselben Konto wie das Container-Image in Amazon ECR können Sie Lambda-Zugriff entweder über die identitätsbasierte Richtlinie für die Ausführungsrolle oder die ressourcenbasierte Richtlinie für das Amazon ECR-Repository gewähren. Nur eine Seite muss den Zugriff zulassen.

Wenn Sie sich für die Verwendung einer Amazon ECR-Repository-Richtlinie entscheiden, fügen Sie `ecr:GetDownloadUrlForLayer` Berechtigungen `ecr:BatchGetImage` hinzu. Das folgende Beispiel zeigt die Mindestrichtlinie:

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
```

```
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

Weitere Informationen zu Amazon-ECR-Repository-Berechtigungen finden Sie unter Private Repository-Richtlinien im Benutzerhandbuch zu Amazon Elastic Container Registry.

Wenn das Amazon-ECR-Repository diese Berechtigungen nicht enthält, versucht Lambda, sie automatisch hinzuzufügen. Lambda kann Berechtigungen nur hinzufügen, wenn der Lambda aufrufende Prinzipal über `ecr:getRepositoryPolicy`- und `ecr:setRepositoryPolicy`-Berechtigungen verfügt.

Um Ihre Repository-Berechtigungen für Amazon ECR anzuzeigen oder zu bearbeiten, befolgen Sie die Anweisungen unter Festlegung einer privaten Repository-Richtlinienanweisung im Benutzerhandbuch zu Amazon Elastic Container Registry.

Kontoübergreifende Berechtigungen von Amazon ECR

Wenn eine Funktion in einem Konto ein Container-Image aus einem Amazon ECR-Repository in einem anderen Konto verwendet, müssen beide Seiten Zugriff gewähren. Die identitätsbasierte Richtlinie für die Rolle im verbrauchenden Konto muss `ecr:BatchGetImage` und zulassen`ecr:GetDownloadUrlForLayer`, und die ressourcenbasierte Richtlinie für das Amazon ECR-Repository im besitzenden Konto muss diese Aktionen ebenfalls zulassen.

Im folgenden Beispiel benötigt die Berechtigungsrichtlinie Ihres Amazon-ECR-Repository die folgenden Anweisungen, um den Zugriff auf Kontonummer 123456789012 zu gewähren.

- CrossAccountPermission— Ermöglicht dem Konto 123456789012 das Erstellen und Aktualisieren von Lambda-Funktionen, die Bilder aus diesem ECR-Repository verwenden.
- LambdaECRImageCrossAccountRetrievalPolicy— Lambda setzt den Status einer Funktion schließlich auf inaktiv, wenn sie über einen längeren Zeitraum nicht aufgerufen wird. Diese Anweisung ist erforderlich, damit Lambda das Container-Image zur Optimierung und Zwischenspeicherung im Namen der Funktion abrufen kann, die 123456789012 besitzt.

Example– Ihrem Repository kontoübergreifende Berechtigungen hinzufügen

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-repository"
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "ArnLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-repository"
    }
  ]
}
```

Um den Zugriff auf mehrere Konten zu gewähren, fügen Sie die Konto-IDs der Liste der Prinzipalen in der `CrossAccountPermission`-Richtlinie hinzu sowie zur Bedingungs-Auswertungsliste der Bedingung in `LambdaECRImageCrossAccountRetrievalPolicy`.

Wenn Sie mit mehreren Konten in einer AWS Organisation arbeiten, empfehlen wir, dass Sie jede Konto-ID in der ECR-Berechtigungsrichtlinie auflisten. Dieser Ansatz entspricht der bewährten AWS Sicherheitsmethode, enge Berechtigungen in IAM-Richtlinien festzulegen.

Zusätzlich zur Amazon ECR-Repository-Richtlinie muss der Benutzer oder die Rolle, die die Funktion erstellt, auch über `GetDownloadUrlForLayer` Berechtigungen in ihrer identitätsbasierten Richtlinie verfügen`BatchGetImage`.

Lebenszyklus der Funktion

Nachdem Sie ein neues oder aktualisiertes Container-Image hochgeladen haben, optimiert Lambda das Image, bevor die Funktion Aufrufe verarbeiten kann. Der Optimierungsprozess kann einige Sekunden dauern. Die Funktion verbleibt im Zustand `Pending`, bis der Prozess abgeschlossen ist, dann geht der Zustand in `Active` über. Sie können die Funktion erst dann aufrufen, wenn sie den `Active`-Zustand erreicht hat.

Wenn eine Funktion mehrere Wochen lang nicht aufgerufen wird, gewinnt Lambda seine optimierte Version zurück und die Funktion wechselt in den `Inactive`-Zustand. Um die Funktion wieder zu aktivieren, müssen Sie sie aufrufen. Lambda lehnt den ersten Aufruf ab und die Funktion tritt in den `Pending`-Zustand, bis Lambda das Image neu optimiert. Die Funktion kehrt dann zum `Active`-Zustand zurück.

Lambda ruft regelmäßig das zugehörige Container-Image aus dem Amazon ECR Repository ab. Wenn das entsprechende Container-Image in Amazon ECR nicht mehr vorhanden ist oder Berechtigungen widerrufen werden, wechselt die Funktion in den `Failed`-Zustand und Lambda gibt einen Fehler für alle Funktionsaufrufe zurück.

Sie können die Lambda-API verwenden, um Informationen über den Zustand einer Funktion abzurufen. Weitere Informationen finden Sie unter Status der Lambda-Ereignisfunktion.

Self-managed S3-Codespeicher

Wenn Sie eine Lambda-Ereignisfunktion oder -Layer aus einem ZIP-Dateiarchiv erstellen oder aktualisieren, speichert Lambda standardmäßig eine Kopie Ihres Codes im Speicher. Lambda-managed Jedes AWS Konto hat ein Lambda-managed Speicherkontingent von 300 GB pro Region.

Mit dem selbstverwalteten S3-Codespeicher können Sie Lambda so konfigurieren, dass Ihr Code direkt aus einem S3-Bucket in Ihrem Konto referenziert wird. Lambda speichert keine Kopie Ihres Codes, sodass der Code nicht auf Ihr Lambda-managed Speicherkontingent angerechnet wird. Lambda greift direkt von Ihrem S3-Bucket aus auf Ihren Code zu.

Self-managed Der S3-Codespeicher ist für alle Funktionen und Ebenen verfügbar, die mithilfe von ZIP-Dateiarchiven erstellt und aktualisiert wurden. Es ist mit allen S3-Speicherklassen außer Glacier-Speicherklassen kompatibel.

> **Note**
>
> Self-managed Der S3-Codespeicher ändert nichts an der maximalen Größe des ZIP-Bereitstellungspakets, die weiterhin bei 250 MB (entpackt) liegt.

Themen

- Einrichtung eines selbstverwalteten S3-Codespeichers
- Konfigurieren von Berechtigungen
- Eine Funktion mit selbstverwaltetem S3-Speicher erstellen
- Verwenden von selbstverwaltetem S3-Speicher mit Ebenen
- Zwischen Speichermodi wechseln
- Funktionslebenszyklus mit selbstverwaltetem Speicher
- Cross-account und regionsübergreifende Nutzung
- Überlegungen

Einrichtung eines selbstverwalteten S3-Codespeichers

Gehen Sie wie folgt vor, um den selbstverwalteten S3-Codespeicher zu verwenden:

1. Erstellen Sie einen S3-Bucket oder verwenden Sie einen vorhandenen Bucket in Ihrem Konto.

2. Aktivieren Sie die S3-Versionierung für den Bucket. Lambda benötigt S3-Versionierung, um zu verfolgen, welche Version Ihres Quellobjekts verwendet werden soll. Jedes Mal, wenn Sie ein neues .zip-Paket hochladen, erstellt S3 eine neue Version, sodass Lambda Aktualisierungen unterscheiden kann.
3. Laden Sie Ihr .zip-Bereitstellungspaket in den S3-Bucket hoch.
4. Erteilen Sie dem Lambda-Dienstprinzipal `s3:GetObject` und `s3:GetObjectVersion` Berechtigungen für das Quellobjekt. Siehe Berechtigungen konfigurieren.
5. Erstellen oder aktualisieren Sie Ihre Funktion oder Ebene, indem Sie das S3-Objekt, den S3-Schlüssel, die S3-Objektversion und die Einstellung `S3ObjectStorageMode` auf angeben`REFERENCE`.

Konfigurieren von Berechtigungen

Um den selbstverwalteten S3-Codespeicher zu verwenden, gewähren Sie dem Lambda Service Principal (`lambda.amazonaws.com`) die Berechtigung, auf Ihre Quellobjekte zuzugreifen. Fügen Sie die folgende S3-Bucket-Richtlinie hinzu:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "LambdaSelfManagedCodeAccess",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion"
      ],
      "Resource": [
        "arn:aws:s3:::my-bucket/my-function.zip"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:lambda:us-east-1:111122223333:function:my-function"
        }
      }
    }
  ]
```

```
}
```

Diese S3-Bucket-Richtlinie ist unabhängig von der Ausführungsrolle Ihrer Funktion. Die Bucket-Richtlinie ist eine ressourcenbasierte Richtlinie, die dem Lambda Service Principal (`lambda.amazonaws.com`) die Berechtigung erteilt, ein bestimmtes Codeobjekt abzurufen, sodass Lambda Ihre Funktion optimieren kann. Sie erteilt keinem anderen Prinzipal Berechtigungen und gewährt auch keine Erlaubnis, Ihre Funktion aufzurufen. Die Ausführungsrolle Ihrer Funktion ist eine separate IAM-Identität, die Lambda annimmt, wenn es Ihre Funktion ausführt. Sie gewähren Aufruf- und andere Laufzeitberechtigungen über IAM-Richtlinien, die dieser Rolle zugeordnet sind. Wenn Sie dem Lambda-Serviceprinzipal Zugriff auf Objekte in Ihrem Bucket gewähren, können andere Identitäten nicht auf diese Objekte zugreifen oder Ihre Funktionen aufrufen.

Eine Funktion mit selbstverwaltetem S3-Speicher erstellen

Sie können eine Funktion mit selbstverwaltetem S3-Codespeicher mithilfe der Konsole AWS CLI, der oder erstellen. AWS CloudFormation

> **Note**
>
> Der Code-Editor der Lambda-Konsole ist für Funktionen, die den Referenzmodus verwenden, nicht verfügbar. Um den Funktionscode zu bearbeiten, aktualisieren Sie die ZIP-Datei in Ihrem S3-Bucket und aktualisieren Sie die Funktion.

Verwenden der Konsole

Wenn Sie eine Funktion von einem S3-Standort in der Lambda-Konsole aus aktualisieren, können Sie den Codespeichermodus wählen.

Um eine Funktion zu aktualisieren, um den Referenzmodus (Konsole) zu verwenden

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Wählen Sie unter Codequelle die Option Update und dann Update aus einer Datei in Amazon S3 aus.
4. Geben Sie für Amazon S3 S3-Link-URL die S3-Link-URL Ihrer ZIP-Datei ein.
5. Wählen Sie unter Code-Speichermodus die Option Referenzmodus aus.
6. Wählen Sie Speichern.

Nachdem Sie den Referenzmodus konfiguriert haben, werden im Abschnitt Codeeigenschaften auf der Registerkarte Code der Funktion der Codespeichermodus: Referenz und der S3-Speicherort angezeigt, der auf Ihren S3-Bucket verweist.

Verwendung der AWS CLI

Verwenden Sie den `create-function` Befehl mit dem `S3ObjectStorageMode=REFERENCE` Parameter:

```
aws lambda create-function \
  --function-name my-function \
  --runtime python3.12 \
  --role arn:aws:iam::111122223333:role/lambda-execution-role \
  --handler lambda_function.lambda_handler \
  --code S3Bucket=my-bucket,\
S3Key=my-function.zip,\
S3ObjectVersion=abc123def456,\
S3ObjectStorageMode=REFERENCE
```

Um eine bestehende Funktion zu aktualisieren, um selbstverwalteten S3-Speicher zu verwenden:

```
aws lambda update-function-code \
  --function-name my-function \
  --s3-bucket my-bucket \
  --s3-key my-function.zip \
  --s3-object-version abc123def456 \
  --s3-object-storage-mode REFERENCE
```

> **Important**
>
> Sie müssen `S3ObjectStorageMode=REFERENCE` bei jedem Aufruf angeben. `update-function-code` Wenn Sie es weglassen`S3ObjectStorageMode`, ist es standardmäßig `COPY` und Lambda speichert Ihren Code im Speicher. Lambda-managed

Verwenden AWS CloudFormation

Stellen `S3ObjectStorageMode` Sie in Ihrer CloudFormation Vorlage `REFERENCE` in der `Code` Eigenschaft der Ressource Folgendes ein: `AWS::Lambda::Function`

```
Resources:
```

```
  MyFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-function
      Runtime: python3.12
      Handler: lambda_function.lambda_handler
      Role: !GetAtt LambdaExecutionRole.Arn
      Code:
        S3Bucket: my-bucket
        S3Key: my-function.zip
        S3ObjectVersion: abc123def456
        S3ObjectStorageMode: REFERENCE
```

Verwenden von selbstverwaltetem S3-Speicher mit Ebenen

Sie können auch den selbstverwalteten S3-Codespeicher für Layer-Versionen verwenden. Verwenden Sie den `publish-layer-version` Befehl mit`S3ObjectStorageMode=REFERENCE`:

```
aws lambda publish-layer-version \
  --layer-name my-layer \
  --content S3Bucket=my-bucket,S3Key=my-layer.zip,S3ObjectStorageMode=REFERENCE \
  --compatible-runtimes python3.12
```

Sie können eine beliebige Kombination aus Lambda-managed selbstverwaltetem Speicher für Funktionscode und Layercode verwenden.

Zwischen Speichermodi wechseln

Sie können zwischen dem Lambda-managed selbstverwalteten S3-Codespeicher wechseln, wenn Sie Ihren Funktionscode aktualisieren oder eine neue Layer-Version veröffentlichen. Stellen `S3ObjectStorageMode` Sie entweder `COPY` oder `REFERENCE` ein:

- `COPY`(Standard) — Lambda kopiert Ihren Quellcode in den Lambda-managed Speicher.
- `REFERENCE`— Lambda referenziert Ihren Quellcode direkt aus Ihrem S3-Bucket, ohne eine Kopie zu speichern.

Wenn Sie von `COPY` zu wechseln`REFERENCE`, löscht Lambda die Lambda-managed Kopie Ihres Quellcodes. Wenn Sie von `REFERENCE` zu wechseln`COPY`, erstellt Lambda eine Kopie Ihres Quellcodes im Lambda-managed Speicher. Der Wechsel zu `COPY` schlägt fehl, wenn Sie das Lambda-managed Code-Speicherlimit überschreiten.

Funktionslebenszyklus mit selbstverwaltetem Speicher

Lambda greift regelmäßig auf das Quellobjekt aus Ihrem S3-Bucket zu, um Ihren Funktionscode neu zu optimieren. Sie müssen den Zugriff auf das Quellobjekt aufrechterhalten, damit Ihre Funktion aktiv bleibt.

- Wenn Lambda den Zugriff auf das Quellobjekt für eine Funktion verliert, wechselt die Funktion in den `Inactive` Status. Um die Funktion wiederherzustellen, stellen Sie den Zugriff auf das Quellobjekt wieder her und aktualisieren Sie die Funktion.
- Wenn Lambda den Zugriff auf das Quellobjekt für eine Ebene verliert, bleibt die Funktion erhaltenActive. Sie können die Funktionskonfiguration aktualisieren (ohne die Ebenen zu aktualisieren), den Funktionscode aktualisieren oder die Funktionskonfiguration mit neuen Ebenen aktualisieren. Das Aktualisieren der Funktionskonfiguration mit der Ebene, auf die nicht zugegriffen werden kann, schlägt jedoch fehl.

Cross-account und regionsübergreifende Nutzung

Sie können Funktionen und Ebenen aus dem Quellcode in S3-Buckets in verschiedenen Konten und in verschiedenen Regionen erstellen. Cross-Region Beim Abrufen von Objekten fallen zusätzliche Kosten für die S3-Datenübertragung an. Einzelheiten finden Sie unter S3-Preise.

Überlegungen

- Self-managed Der S3-Codespeicher ist nur für Archivfunktionen und Ebenen von ZIP-Dateien verfügbar. Container-Image-Funktionen verwenden weiterhin Amazon ECR.
- Die maximale Größe des ZIP-Bereitstellungspakets (250 MB entpackt) ist unverändert.
- Sie zahlen die standardmäßigen Amazon S3 S3-Speichergebühren für Code, der in Ihren selbstverwalteten Buckets gespeichert ist. Lambda berechnet keine Gebühren für den Abruf von S3-Objekten innerhalb derselben Region.
- Self-managed Der S3-Codespeicher ist mit allen S3-Speicherklassen außer Glacier-Speicherklassen kompatibel.
- Sie können S3-Funktionen wie die Cross-Region S3-Replikation verwenden, um Artefakte regionsübergreifend gemeinsam zu nutzen, und S3-Lebenszyklusrichtlinien zur Verwaltung von Objekten verwenden.

Speicher der Lambda-Ereignisfunktion konfigurieren

Lambda weist die Rechenleistung proportional zur Menge des konfigurierten Arbeitsspeichers zu. Arbeitsspeicher ist die Menge an Arbeitsspeicher, die Ihrer Lambda-Funktion zur Laufzeit zur Verfügung steht. Sie können den Arbeitsspeicher und die CPU-Leistung, die Ihrer Funktion zugewiesen sind, mit der Einstellung Arbeitsspeicher erhöhen oder verringern. Sie können Speicher zwischen 128 MB und 10 240 MB in 1-MB-Schritten konfigurieren. Bei 1.769 MB entspricht eine Funktion einer vCPU (ein V Credits pro CPU-second Sekunde).

Auf dieser Seite wird beschrieben, wie und wann die Speichereinstellung für eine Lambda-Funktion aktualisiert werden muss.

Sections

- Ermitteln der geeigneten Speichereinstellung für eine Lambda-Funktion
- Konfigurieren des Funktionsspeichers (Konsole)
- Konfiguration des Funktionsspeichers (AWS CLI)
- Konfiguration des Funktionsspeichers (AWS SAM)
- Akzeptieren von Empfehlungen für den Funktionsspeicher (Konsole)

Ermitteln der geeigneten Speichereinstellung für eine Lambda-Funktion

Der Speicher ist der wichtigste Hebel zur Steuerung der Ausführung einer Funktion. Die Standardeinstellung, 128 MB, ist die niedrigste mögliche Einstellung. Wir empfehlen, nur 128 MB für einfache Lambda-Funktionen zu verwenden, z. B. solche, die Ereignisse transformieren und an andere AWS -Dienste weiterleiten.

Eine höhere Speicherzuweisung kann die Leistung von Funktionen verbessern, die importierte Bibliotheken, Lambda-Ebenen, Amazon Simple Storage Service (Amazon S3) oder Amazon Elastic File System (Amazon EFS) verwenden. Durch die Erweiterung des Arbeitsspeichers erhöht sich proportional die Anzahl der CPUs, wodurch die verfügbare Rechenleistung insgesamt steigt. Wenn eine Funktion an die CPU, das Netzwerk oder den Speicher gebunden ist, kann eine Erhöhung der Speichereinstellung die Leistung erheblich verbessern.

Um die richtige Speicherkonfiguration zu finden, überwachen Sie Ihre Funktionen mit Amazon CloudWatch und stellen Sie Alarme ein, wenn sich der Speicherverbrauch den konfigurierten Höchstwerten nähert. Dies kann helfen, speichergebundene Funktionen zu identifizieren. Für

CPU-bound und IO-bound Funktionen kann die Überwachung der Dauer ebenfalls Aufschluss geben. In diesen Fällen kann eine Erhöhung des Speichers dazu beitragen, die Rechen- oder Netzwerkengpässe zu beheben.

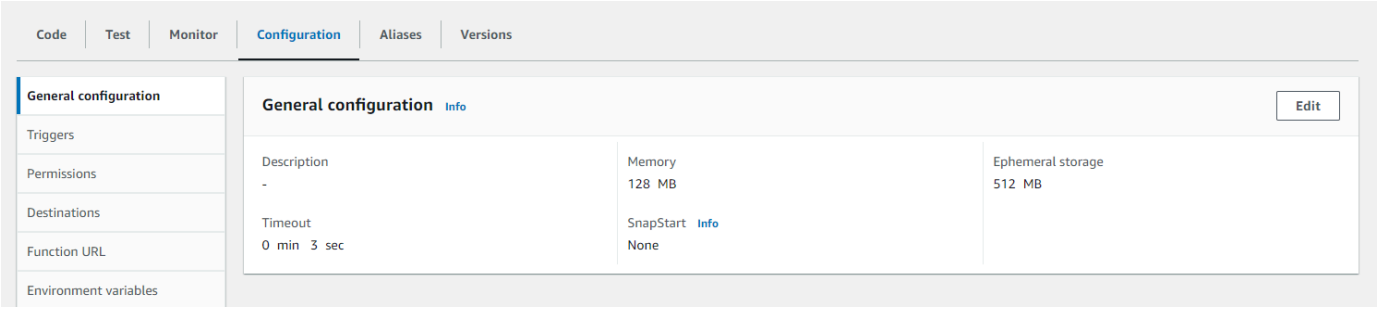
Sie können auch die Verwendung des Open-Source-Tools AWS Lambda Power Tuning in Betracht ziehen. Dieses Tool verwendet AWS Step Functions , um mehrere gleichzeitige Versionen einer Lambda-Funktion an unterschiedlichen Speicherzuweisungen auszuführen und die Leistung zu messen. Die Eingabefunktion wird in Ihrem AWS Konto ausgeführt und führt Live-HTTP-Aufrufe und SDK-Interaktionen durch, um die wahrscheinliche Leistung in einem Live-Produktionsszenario zu messen. Sie können auch einen CI/CD Prozess implementieren, um dieses Tool zur automatischen Messung der Leistung neuer Funktionen zu verwenden, die Sie bereitstellen.

Konfigurieren des Funktionsspeichers (Konsole)

Sie können den Speicher Ihrer Funktion in der Lambda-Konsole konfigurieren.

So aktualisieren Sie den Speicher einer Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).



4. Wählen Sie unter General configuration (allgemeine Konfiguration) Bearbeiten.
5. Legen Sie für Arbeitsspeicher einen Wert von 128 MB bis 10.240 MB fest.
6. Wählen Sie Speichern.

Konfiguration des Funktionsspeichers (AWS CLI)

Sie können den Befehl update-function-configuration verwenden, um den Speicher Ihrer Funktion zu konfigurieren.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

Konfiguration des Funktionsspeichers (AWS SAM)

Sie können den AWS Serverless Application Model verwenden, um den Speicher für Ihre Funktion zu konfigurieren. Aktualisieren Sie die MemorySizeEigenschaft in Ihrer `template.yaml` Datei und führen Sie dann sam deploy aus.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
      # Other function properties...
```

Akzeptieren von Empfehlungen für den Funktionsspeicher (Konsole)

Wenn Sie Administratorrechte in AWS Identity and Access Management (IAM) haben, können Sie sich dafür entscheiden, Empfehlungen für Lambda-Funktionsspeichereinstellungen von zu erhalten. AWS Compute Optimizer Anweisungen dazu, wie Sie Speicherempfehlungen für Ihr Konto oder Ihr Unternehmen aktivieren, finden Sie unter Aktivieren Ihres Kontos im AWS Compute Optimizer - Benutzerhandbuch.

> **Note**
>
> Compute Optimizer unterstützt nur Funktionen, die die x86_64-Architektur verwenden.

Wenn Sie sich angemeldet haben und Ihre Lambda-Funktion die Anforderungen von Compute Optimizer erfüllt, können Sie Empfehlungen zum Funktionsspeicher von Compute Optimizer in der Lambda-Konsole unter Allgemeine Konfiguration anzeigen und akzeptieren.

Konfigurieren Sie kurzlebigen Speicher für Lambda-Ereignisfunktionen

Lambda bietet flüchtigen Speicher für Funktionen im `/tmp`-Verzeichnis. Dieser Speicher ist temporär und für jede Ausführungsumgebung einzigartig. Sie können die Menge des flüchtigen Speichers, der Ihrer Funktion zugewiesen ist, mithilfe der Einstellung Flüchtiger Speicher steuern. Sie können den flüchtigen Speicher zwischen 512 MB und 10.240 MB in 1-MB-Schritten konfigurieren. Alle in `/tmp` gespeicherten Daten werden im Ruhezustand mit einem von AWS-verwalteten Schlüssel verschlüsselt.

Auf dieser Seite werden allgemeine Anwendungsfälle und die Aktualisierung des kurzlebigen Speichers für eine Lambda-Funktion beschrieben.

Sections

- Häufige Anwendungsfälle für flüchtiger Speicher
- Flüchtigen Speicher konfigurieren (Konsole)
- Konfiguration kurzlebigen Speichers (AWS CLI)
- Konfiguration kurzlebigen Speichers (AWS SAM)

Häufige Anwendungsfälle für flüchtiger Speicher

Im Folgenden sind einige häufig vorkommende Anwendungsfälle aufgeführt, die von einem erhöhten flüchtigen Speicher profitieren:

- Extract-transform-load (ETL-) Jobs: Erhöhen Sie den temporären Speicher, wenn Ihr Code Zwischenberechnungen durchführt oder andere Ressourcen herunterlädt, um die Verarbeitung abzuschließen. Mehr temporärer Speicherplatz ermöglicht die Ausführung komplexerer ETL-Jobs in Lambda-Funktionen.
- Inferenz durch Machine Learning(ML): Viele Inferenzaufgaben basieren auf großen Referenzdatendateien, einschließlich Bibliotheken und Modellen. Mit mehr ephemerem Speicher können Sie größere Modelle vom Amazon Simple Storage Service (Amazon S3) auf `/tmp` herunterladen und in Ihrer Verarbeitung verwenden.
- Datenverarbeitung: Bei Workloads, die Objekte von Amazon S3 als Reaktion auf S3-Ereignisse herunterladen, ermöglicht mehr `/tmp`-Speicherplatz die Verarbeitung größerer Objekte ohne In-Memory-Verarbeitung. Workloads, die PDF-Dateien erstellen oder Medien verarbeiten, profitieren ebenfalls von einer kurzlebigen Speicherung.

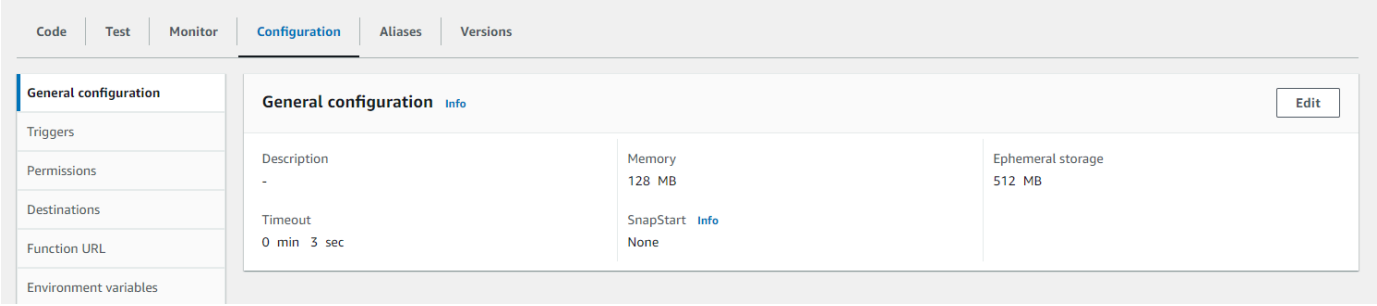
- Grafikverarbeitung: Die Bildverarbeitung ist ein häufiger Anwendungsfall für Anwendungen. Lambda-based Bei Workloads, die große TIFF-Dateien oder Satellitenbilder verarbeiten, erleichtert ein kurzlebigerer Speicher die Verwendung von Bibliotheken und die Durchführung der Berechnung in Lambda.

Flüchtigen Speicher konfigurieren (Konsole)

Flüchtiger Speicher können Sie in der Lambda-Konsole konfigurieren.

So ändern Sie den flüchtigen Speicher für eine Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).



4. Wählen Sie unter General configuration (allgemeine Konfiguration) Bearbeiten.
5. Legen Sie für den flüchtigen Speicher einen Wert zwischen 512 MB und 10.240 MB fest, in 1-MB-Schritten.
6. Wählen Sie Speichern.

Konfiguration kurzlebigen Speichers (AWS CLI)

Sie können den Befehl update-function-configuration verwenden, um den flüchtigen Speicher zu konfigurieren.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

Konfiguration kurzlebigen Speichers (AWS SAM)

Sie können den AWS Serverless Application Model verwenden, um den flüchtigen Speicher für Ihre Funktion zu konfigurieren. Aktualisieren Sie die EphemeralStorageEigenschaft in Ihrer `template.yaml` Datei und führen Sie dann sam deploy aus.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs22.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

Auswahl und Konfiguration einer Befehlssatzarchitektur für Ihre Lambda-Funktion

Die-Befehlssatz-Architektureiner Lambda-Funktion bestimmt den Typ des Computerprozessors, den Lambda zum Ausführen der Funktion verwendet. Lambda bietet eine Auswahl an Befehlssatz-Architekturen:

- arm64 — 64-Bit-ARM-Architektur, für AWS Graviton-Prozessoren.
- x86_64 — 64-Bit-x86-Architektur für x86-basierte Prozessoren.

> **Note**
>
> Die arm64-Architektur ist in den meisten verfügbar AWS-Regionen. Weitere Informationen finden Sie unter AWS Lambda – Preise. Wählen Sie in der Speicherpreistabelle die Preis von Arm Tab, und öffnen Sie dann Region Drop-down-Liste, um zu sehen, welche AWS-Regionen unterstütze arm64 mit Lambda.
> Ein Beispiel für die Erstellung einer Funktion mit der Arm64-Architektur finden Sie unter AWS Lambda Funktionen Bereitgestellt von AWS Graviton2-Prozessor.

Themen

- Vorteile der Verwendung von arm64-Architektur
- Anforderungen für die Migration zur arm64-Architektur
- Funktionscode-Kompatibilität mit arm64-Architektur
- Migration zur arm64-Architektur
- Konfigurieren der Befehlssatz-Architektur

Vorteile der Verwendung von arm64-Architektur

Lambda-Funktionen, die die arm64-Architektur verwenden (AWS Graviton-Prozessoren) können ein deutlich besseres Preis und eine bessere Leistung erzielen als die entsprechende Funktion, die auf der x86_64-Architektur ausgeführt wird. Erwägen Sie, arm64 für rechenintensive Anwendungen wie Hochleistungsrechnen, Videocodierung und Simulations-Workloads zu verwenden.

Lambda verwaltet die zugrunde liegende Hardware und garantiert keine bestimmte Prozessorgeneration: Ihre Funktion kann je nach regionaler Hardwareverfügbarkeit auf

verschiedenen Generationen des Graviton-Prozessors ausgeführt werden. Ihr Code zielt nicht auf einen bestimmten Chipsatz ab; er muss nur mit der Arm64-Architektur (AArch64) kompatibel sein. Alle Graviton-Generationen bieten diese Vorteile gegenüber x86_64:

- Ein größerer L2-Cache pro vCPU, der die Speicherlesezeit reduziert und die Latenz für Web- und Mobil-Backends, Microservices und Datenverarbeitungssysteme verbessert.
- Verbesserte Verschlüsselungsleistung durch kryptografische ARM-Erweiterungen.
- Unterstützung für Befehlssätze, die die Latenz von verringern CPU-based Inferenz für maschinelles Lernen.

Weitere Informationen finden Sie unter AWS Graviton-Prozessoren.

Anforderungen für die Migration zur arm64-Architektur

Wenn Sie eine Lambda-Funktion für die Migration zur arm64-Architektur auswählen, stellen Sie sicher, dass Ihre Funktion die folgenden Anforderungen erfüllt, um eine reibungslose Migration zu gewährleisten:

- Das Bereitstellungspaket enthält nur Open-Source-Komponenten und Quellcode, die Sie steuern, sodass Sie alle notwendigen Updates für die Migration vornehmen können.
- Wenn der Funktionscode Abhängigkeiten von Drittanbietern enthält, bietet jede Bibliothek oder jedes Paket eine arm64-Version.

Funktionscode-Kompatibilität mit arm64-Architektur

Ihr Lambda-Funktionscode muss mit der Befehlssatz-Architektur der Funktion kompatibel sein. Bevor Sie eine Funktion zur arm64-Architektur migrieren, beachten Sie die folgenden Punkte zum aktuellen Funktionscode:

- Wenn Sie Ihren Funktionscode mit dem eingebetteten Code-Editor hinzugefügt haben, läuft Ihr Code wahrscheinlich auf beiden Architekturen ohne Änderung.
- Nach dem Upload Ihres Funktionscodes, müssen Sie neuen Code uploaden, der mit Ihrer Zielarchitektur kompatibel ist.
- Wenn Ihre Funktion Layer verwendet, müssen Sieüberprüfe jede Ebeneum sicherzustellen, dass es mit der neuen Architektur kompatibel ist. Wenn ein Layer nicht kompatibel ist, bearbeiten Sie die Funktion, um die aktuelle Layer-Version durch eine kompatible Layer-Version zu ersetzen.

- Wenn Ihre Funktion Lambda-Erweiterungen verwendet, müssen Sie jede Erweiterung überprüfen, um sicherzustellen, dass sie mit der neuen Architektur kompatibel ist.
- Wenn Ihre Funktion einen Container-Image-Bereitstellungspakettyp verwendet, müssen Sie ein neues Container-Image erstellen, das mit der Architektur der Funktion kompatibel ist.

Migration zur arm64-Architektur

Um eine Lambda-Funktion auf die arm64-Architektur zu migrieren, empfehlen wir die folgenden Schritte:

1. Erstellen Sie die Liste der Abhängigkeiten für Ihre Anwendung oder Ihren Workload. Häufige Abhängigkeiten sind unter anderem:
   - Alle Bibliotheken und Pakete, die die Funktion verwendet.
   - Die Tools, die Sie zum Erstellen, Bereitstellen und Testen der Funktion verwenden, z. B. Compiler, Testsuiten, Continuous Integration und Continuous Delivery (CI/CD) Pipelines, Bereitstellungstools und Skripts.
   - Die Lambda-Erweiterungen und Tools von Drittanbietern, mit denen Sie die Funktion in der Produktion überwachen.
2. Überprüfen Sie für jede der Abhängigkeiten die Version und prüfen Sie dann, ob arm64-Versionen verfügbar sind.
3. Erstellen Sie eine Umgebung, um Ihre Anwendung zu migrieren.
4. Bootstrap die Anwendung.
5. Testen und debuggen Sie die Anwendung.
6. Testen Sie die Leistung der arm64-Funktion. Vergleichen Sie die Leistung mit der x86_64-Version.
7. Aktualisieren Sie Ihre Infrastruktur-Pipeline, um arm64 Lambda-Funktionen zu unterstützen.
8. Stufen Sie Ihre Bereitstellung in die Produktion ein.

   Verwenden Sie zum Beispiel Alias-Routing-Konfiguration um den Datenverkehr zwischen den Versionen x86 und arm64 der Funktion aufzuteilen und die Leistung und Latenz zu vergleichen.

Weitere Informationen zum Erstellen einer Codeumgebung für die Arm64-Architektur, einschließlich sprachspezifischer Informationen für Java, Go, .NET und Python, finden Sie im Erste Schritte mit AWS Graviton GitHub Endlager.

Konfigurieren der Befehlssatz-Architektur

Sie können die Befehlssatzarchitektur für neue und bestehende Lambda-Funktionen mithilfe der Lambda-Konsole konfigurieren. AWS SDKs, AWS Command Line Interface (AWS CLI), oder CloudFormation. Führen Sie die folgenden Schritte aus, um die Befehlssatzarchitektur für eine vorhandene Lambda-Funktion von der Konsole aus zu ändern.

1. Öffne das Seite „Event-Funktionen“ der Lambda-Konsole.
2. Klicken Sie auf den Namen der Funktion, für die Sie die Befehlssatzarchitektur konfigurieren möchten.
3. Wählen Sie auf der Registerkarte Code für den Abschnitt Laufzeiteinstellungen die Option Bearbeiten aus.
4. Unter Architecture (Architektur) wählen Sie die Befehlssatzarchitektur aus, die Ihre Funktion verwenden soll.
5. Wählen Sie Speichern.

Timeout für die Lambda-Ereignisfunktion konfigurieren

Lambda führt Ihren Code für eine festgelegte Zeitspanne aus, bevor ein Timeout erfolgt. Beim Timeout handelt es sich um die maximale Zeitspanne in Sekunden, die eine Lambda-Funktion ausgeführt werden kann. Der Standardwert für diese Einstellung ist 3 Sekunden. Sie können ihn jedoch in Schritten von 1 Sekunde bis zu einem Höchstwert von 900 Sekunden (15 Minuten) anpassen.

Für Funktionen, die AWS Lambda Managed Instances verwenden, unterstützen asynchrone Aufrufe und Aufrufe zur Zuordnung von Ereignisquellen (außer Amazon MQ und Amazon DocumentDB) einen zulässigen Höchstwert von 5.400 Sekunden (90 Minuten).

Auf dieser Seite wird beschrieben, wie und wann die Timeout-Einstellung für eine Lambda-Funktion aktualisiert werden muss.

Sections

- Ermitteln des geeigneten Timeout-Werts für eine Lambda-Funktion
- Konfigurieren des Timeouts (Konsole)
- Timeout konfigurieren (AWS CLI)
- Timeout konfigurieren (AWS SAM)

Ermitteln des geeigneten Timeout-Werts für eine Lambda-Funktion

Wenn der Timeout-Wert nahe an der durchschnittlichen Dauer einer Funktion liegt, besteht ein höheres Risiko, dass bei der Funktion ein unerwarteter Timeout auftritt. Die Dauer einer Funktion kann je nach Umfang der Datenübertragung und -verarbeitung sowie der Latenz aller Dienste, mit denen die Funktion interagiert, variieren. Zu den häufigsten Ursachen für Timeouts gehören:

- Downloads von Amazon Simple Storage Service (Amazon S3) sind größer oder dauern länger als der Durchschnitt.
- Eine Funktion sendet eine Anfrage an einen anderen Service, dessen Beantwortung länger dauert.
- Die einer Funktion zur Verfügung gestellten Parameter erfordern eine höhere Rechenkomplexität in der Funktion, wodurch der Aufruf länger dauert.

Stellen Sie beim Testen Ihrer Anwendung sicher, dass Ihre Tests die Größe und Menge der Daten sowie realistische Parameterwerte genau wiedergeben. Tests verwenden aus Gründen der

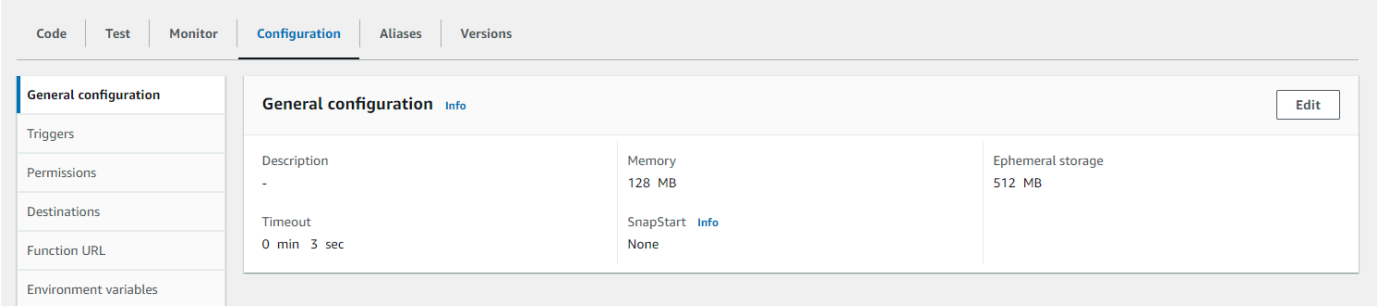
Benutzerfreundlichkeit häufig kleine Stichproben. Sie sollten jedoch Datensätze verwenden, die sich an der Obergrenze dessen befinden, was für Ihren Workload vernünftigerweise zu erwarten ist.

Konfigurieren des Timeouts (Konsole)

Sie können das Funktions-Timeout in der Lambda-Konsole konfigurieren.

So ändern Sie das Timeout für eine Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).



4. Wählen Sie unter General configuration (allgemeine Konfiguration) Bearbeiten.
5. Für Timeout stellen Sie einen Wert zwischen 1 und 900 Sekunden (15 Minuten) ein.
6. Wählen Sie Speichern.

Timeout konfigurieren (AWS CLI)

Sie können den Befehl update-function-configuration verwenden, um den Timeout-Wert in Sekunden zu konfigurieren. Der folgende Beispielbefehl erhöht das Funktions-Timeout auf 120 Sekunden (2 Minuten).

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

Timeout konfigurieren (AWS SAM)

Sie können den AWS Serverless Application Model verwenden, um den Timeout-Wert in Sekunden zu konfigurieren. Aktualisieren Sie die Eigenschaft Timeout in Ihrer `template.yaml`-Datei und führen Sie dann sam deploy aus.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      # Other function properties...
```

Arbeiten mit Lambda-Umgebungsvariablen

Sie können Umgebungsvariablen verwenden, um das Verhalten Ihrer Funktion anzupassen, ohne Code zu aktualisieren. Eine Umgebungsvariable ist ein Zeichenfolgenpaar, das in der versionsspezifischen Konfiguration einer Funktion gespeichert ist. Die Lambda-Laufzeit stellt Umgebungsvariablen für den Code zur Verfügung und legt zusätzliche Umgebungsvariablen mit Informationen über die Funktion und die Aufrufanforderung fest.

> **Note**
>
> Um die Sicherheit zu erhöhen, empfehlen wir, AWS Secrets Manager anstelle von Umgebungsvariablen Datenbankanmeldedaten und anderen vertraulichen Informationen wie API-Schlüsseln oder Autorisierungstoken zu speichern. Weitere Informationen finden Sie unter Verwenden Sie Secrets Manager Manager-Geheimnisse in Lambda-Ereignisfunktionen.

Umgebungsvariablen werden vor dem Funktionsaufruf nicht ausgewertet. Jeder von Ihnen definierte Wert wird als Literalzeichenfolge betrachtet und nicht erweitert. Führen Sie die Variablenauswertung in Ihrem Funktionscode durch.

Erstellen von Lambda-Umgebungsvariablen

Sie können Umgebungsvariablen in Lambda mithilfe der Lambda-Konsole, der AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (AWS SAM) oder mithilfe eines AWS SDK konfigurieren.

Console

Sie definieren Umgebungsvariablen für die unveröffentlichte Version Ihrer Funktion. Wenn Sie eine Version veröffentlichen, werden die Umgebungsvariablen für diese Version zusammen mit anderen versionsspezifischen Konfigurationseinstellungen gesperrt.

Sie erstellen eine Umgebungsvariable für Ihre Funktion, indem Sie einen Schlüssel und einen Wert definieren. Ihre Funktion verwendet den Namen des Schlüssels, um den Wert der Umgebungsvariablen abzurufen.

So legen Sie Umgebungsvariablen in der Lambda-Konsole fest

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.

3. Wählen Sie den Tab Configuration (Konfiguration) und dann Environment variables (Umgebungsvariablen) aus.
4. Wählen Sie unter Environment variables (Umgebungsvariablen) die Option Edit (Bearbeiten).
5. Wählen Sie Umgebungsvariablen hinzufügen aus.
6. Geben Sie einen Schlüssel und einen Wert ein.

   Voraussetzungen

   - Schlüssel beginnen mit einem Buchstaben und bestehen aus mindestens zwei Zeichen.
   - Schlüssel enthalten nur Buchstaben, Zahlen und den Unterstrich (_).
   - Schlüssel sind nicht Lambda vorbehalten.
   - Die Gesamtgröße aller Umgebungsvariablen überschreitet nicht 4 KB.
7. Wählen Sie Speichern.

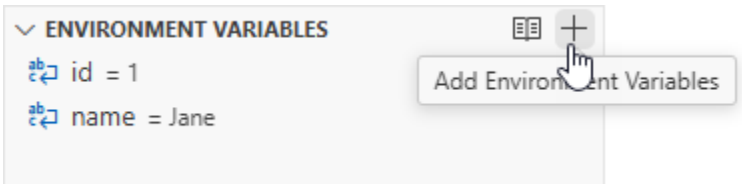
Generieren einer Liste von Umgebungsvariablen im Konsolen-Code-Editor

Sie können im Lambda-Code-Editor eine Liste von Umgebungsvariablen generieren. So können Sie beim Programmieren schnell auf Ihre Umgebungsvariablen verweisen.

1. Wählen Sie die Registerkarte Code.
2. Scrollen Sie im Code-Editor nach unten zum Abschnitt UMGEBUNGSVARIABLEN. Bestehende Umgebungsvariablen sind hier aufgelistet:



3. Um neue Umgebungsvariablen zu erstellen, wählen Sie das Pluszeichen (+):



Umgebungsvariablen bleiben verschlüsselt, wenn sie im Code-Editor der Konsole aufgeführt werden. Wenn Sie Verschlüsselungshilfen für die Verschlüsselung während der Übertragung

aktiviert haben, bleiben diese Einstellungen unverändert. Weitere Informationen finden Sie unter Sicherung von Lambda-Umgebungsvariablen.

Die Liste der Umgebungsvariablen ist schreibgeschützt und nur auf der Lambda-Konsole verfügbar. Diese Datei ist nicht enthalten, wenn Sie das ZIP-Dateiarchiv der Funktion herunterladen, und Sie können keine Umgebungsvariablen hinzufügen, indem Sie diese Datei hochladen.

AWS CLI

Im folgenden Beispiel werden zwei Umgebungsvariablen für eine Funktion mit dem Namen festgeleg `my-function`.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=amzn-s3-demo-bucket,KEY=file.txt}"
```

Wenn Sie Umgebungsvariablen mit dem `update-function-configuration`-Befehl anwenden, wird der gesamte Inhalt der `Variables`-Struktur ersetzt. Um vorhandene Umgebungsvariablen beibehalten, wenn Sie eine neue hinzufügen, schließen Sie alle vorhandenen Werte in Ihre Anforderung ein.

Mit dem `get-function-configuration`-Befehl können Sie die aktuelle Konfiguration abrufen.

```
aws lambda get-function-configuration \
  --function-name my-function
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
```

```
    ...
}
```

Sie können die Revisions-ID aus der Ausgabe von `get-function-configuration` als Parameter an `update-function-configuration` übergeben. Dadurch wird sichergestellt, dass sich die Werte zwischen dem Lesen und dem Aktualisieren der Konfiguration nicht ändern.

Um den Verschlüsselungsschlüssel einer Funktion zu konfigurieren, legen Sie die `KMSKeyARN`-Option fest.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

AWS SAM

Sie können das AWS Serverless Application Model verwenden, um Umgebungsvariablen für Ihre Funktion zu konfigurieren. Aktualisieren Sie die Eigenschaften Umgebung und Variablen in Ihrer `template.yaml`-Datei und führen Sie dann sam deploy aus.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs24.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      Environment:
        Variables:
```

```
        BUCKET: amzn-s3-demo-bucket
        KEY: file.txt
    # Other function properties...
```

AWS SDKs

Verwenden Sie die folgenden API-Operationen, um Umgebungsvariablen mithilfe eines AWS SDK zu verwalten.

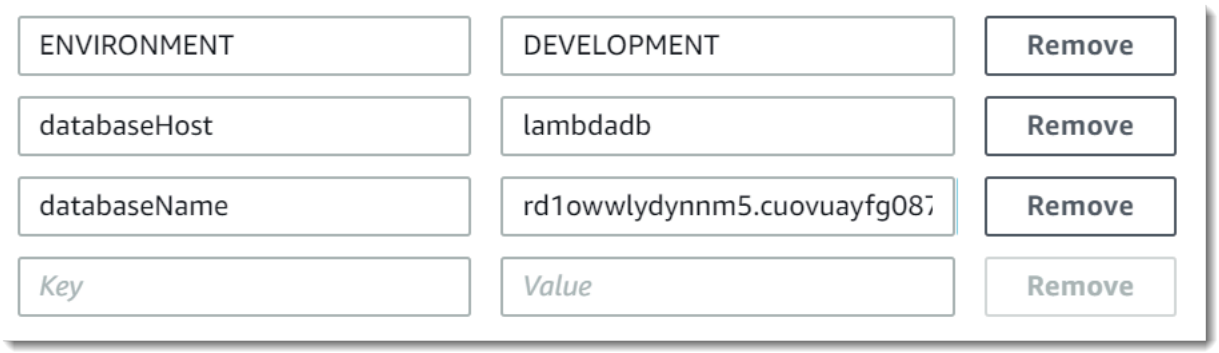
- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Weitere Informationen finden Sie in der AWS -SDK-Dokumentation für Ihre bevorzugte Programmiersprache.

Beispielszenario für Umgebungsvariablen

Sie können Umgebungsvariablen verwenden, um das Funktionsverhalten in Ihrer Testumgebung und Produktionsumgebung anzupassen. Zum Beispiel können Sie zwei Funktionen mit demselben Code aber mit unterschiedlichen Konfigurationen erstellen. Die eine Funktion stellt eine Verbindung mit einer Testdatenbank und die andere mit einer Produktionsdatenbank her. In diesem Fall verwenden Sie Umgebungsvariablen, um den Hostnamen und andere Verbindungsdetails für die Datenbank an die Funktion zu übergeben.

Das folgende Beispiel zeigt, wie der Datenbankhost und der Datenbankname als Umgebungsvariablen definiert werden.



Wenn Ihre Testumgebung mehr Debugging-Informationen generieren soll als die Produktionsumgebung, können Sie eine Umgebungsvariable festlegen, um Ihre Testumgebung so

zu konfigurieren, dass eine ausführlichere Protokollierung oder eine detailliertere Ablaufverfolgung verwendet wird.

Beispielsweise könnten Sie in Ihrer Testumgebung eine Umgebungsvariable mit dem Schlüssel `LOG_LEVEL` und einem Wert festlegen, der die Protokollstufe „Debug“ oder „Trace“ angibt. Im Code Ihrer Lambda-Funktion können Sie dann diese Umgebungsvariable verwenden, um die Protokollstufe festzulegen.

Die folgenden Codebeispiele in Python Node.js veranschaulichen, wie Sie dies erreichen können. In diesen Beispielen wird davon ausgegangen, dass Ihre Umgebungsvariable den Wert `DEBUG` in Python oder `debug` in hat Node.js.

Python

Example Python-Code zum Festlegen der Protokollstufe

```
import os
import logging

# Initialize the logger
logger = logging.getLogger()

# Get the log level from the environment variable and default to INFO if not set
log_level = os.environ.get('LOG_LEVEL', 'INFO')

# Set the log level
logger.setLevel(log_level)

def lambda_handler(event, context):
    # Produce some example log outputs
    logger.debug('This is a log with detailed debug information - shown only in test
 environment')
    logger.info('This is a log with standard information - shown in production and
 test environments')
```

Node.js (ES module format)

Example Node.js Code zum Einstellen der Protokollebene

In diesem Beispiel wird die `winston`-Protokollierungsbibliothek verwendet. Verwenden Sie npm, um diese Bibliothek dem Bereitstellungspaket Ihrer Funktion hinzuzufügen. Weitere Informationen finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen”.

```
import winston from 'winston';

// Initialize the logger using the log level from environment variables, defaulting
 to INFO if not set
const logger = winston.createLogger({
   level: process.env.LOG_LEVEL || 'info',
   format: winston.format.json(),
   transports: [new winston.transports.Console()]
});

export const handler = async (event) => {
   // Produce some example log outputs
   logger.debug('This is a log with detailed debug information - shown only in test
 environment');
   logger.info('This is a log with standard information - shown in production and
 test environment');

};
```

Abrufen von Lambda-Umgebungsvariablen

Umgebungsvariablen in Ihrem Funktionscode können Sie mit der Standardmethode für Ihre Programmiersprache abrufen.

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> In einigen Fällen müssen Sie möglicherweise das folgende Format verwenden:
>
> ```
> region = os.environ.get('AWS_REGION')
> ```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda speichert Umgebungsvariablen sicher, indem sie im Ruhezustand verschlüsselt werden. Sie können Lambda so konfigurieren, dass es einen anderen Verschlüsselungsschlüssel verwendet, Umgebungsvariablenwerte auf der Clientseite verschlüsselt oder Umgebungsvariablen in einer CloudFormation Vorlage mit festlegen. AWS Secrets Manager

Definierte Laufzeitumgebungsvariablen

Lambda Laufzeiten setzen mehrere Umgebungsvariablen während der Initialisierung. Die meisten Umgebungsvariablen stellen Informationen über die Funktion oder Laufzeit bereit. Die Schlüssel für diese Umgebungsvariablen sind reserviert und können nicht in der Funktionskonfiguration gesetzt werden.

Reservierte Umgebungsvariablen

- `_HANDLER` – Der für die Funktion konfigurierte Handler-Speicherort.
- `_X_AMZN_TRACE_ID` – Der X-Ray -Nachverfolgungs-Header. Diese Umgebungsvariable ändert sich bei jedem Aufruf.

  - Diese Umgebungsvariable ist nicht für OS-only Laufzeiten (die `provided` Runtime-Familie) definiert. Sie können `_X_AMZN_TRACE_ID` für benutzerdefinierte Laufzeiten einstellen, indem Sie den Antwort-Header `Lambda-Runtime-Trace-Id` der Nächster Aufruf verwenden.
  - Für die Java-Laufzeit-Versionen 17 und höher wird diese Umgebungsvariable nicht verwendet. Stattdessen speichert Lambda Ablaufverfolgungsinformationen in der Systemeigenschaft `com.amazonaws.xray.traceHeader`.
- `AWS_DEFAULT_REGION`— Die Standardeinstellung AWS-Region , in der die Lambda-Funktion ausgeführt wird.
- `AWS_REGION`— Der AWS-Region Ort, an dem die Lambda-Funktion ausgeführt wird. Wenn eingegeben, überschreibt dieser Wert die `AWS_DEFAULT_REGION`.
  - Weitere Informationen zur Verwendung der AWS-Region Umgebungsvariablen mit AWS SDKs finden Sie unter AWS Region im Referenzhandbuch für AWS SDKs und Tools.
- `AWS_EXECUTION_ENV` – Die Laufzeit-Kennung mit dem Präfix `AWS_Lambda_` z. B. `AWS_Lambda_java8`. Diese Umgebungsvariable ist nicht für OS-only Runtimes (die `provided` Runtime-Familie) definiert.
- `AWS_LAMBDA_FUNCTION_NAME` – Der Name der Funktion.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – Die Menge des für die Funktion verfügbaren Speichers in MB.
- `AWS_LAMBDA_FUNCTION_VERSION` – Die Version der Funktion, die gerade ausgeführt wird.
- `AWS_LAMBDA_INITIALIZATION_TYPE`— Der Initialisierungstyp der Funktion,`on-demand`, `provisioned-concurrency`snap-start, oder. `lambda-managed-instances` Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität,, Verbesserung der Startleistung mit Lambda SnapStart oder. Von Lambda verwaltete Instances
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` — Der Name der Amazon CloudWatch Logs-Gruppe und des Streams für die Funktion. Die `AWS_LAMBDA_LOG_STREAM_NAME` Umgebungsvariablen `AWS_LAMBDA_LOG_GROUP_NAME` und sind in SnapStart Lambda-Funktionen nicht verfügbar.
- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` – Die Zugriffsschlüssel, die über die Ausführungsrolle der Funktion bezogen wurden.
- `AWS_LAMBDA_RUNTIME_API` – (Benutzerdefinierte Laufzeit) Host und Port der Laufzeit-API.
- `LAMBDA_TASK_ROOT` – Der Pfad zu Ihrem Lambda-Funktionscode.
- `LAMBDA_RUNTIME_DIR` – Der Pfad zu Laufzeitbibliotheken.

- `AWS_LAMBDA_MAX_CONCURRENCY`— (Nur Lambda Managed Instances) Die maximale Anzahl gleichzeitiger Aufrufe, die Lambda an eine Ausführungsumgebung sendet.
- `AWS_LAMBDA_METADATA_API`— Die Metadaten-Endpunkt-Serveradresse im Format `{ipv4_address}:{port}` (z. B.). `169.254.100.1:9001`
- `AWS_LAMBDA_METADATA_TOKEN`— Ein eindeutiges Authentifizierungstoken für die aktuelle Ausführungsumgebung, das zur Authentifizierung von Anfragen an den Metadaten-Endpunkt verwendet wird. Lambda generiert dieses Token automatisch bei der Initialisierung.

Die folgenden zusätzlichen Umgebungsvariablen sind nicht vorbehalten und können in Ihrer Funktionskonfiguration erweitert werden.

Nicht reservierte Umgebungsvariablen

- `LANG` – Das Gebietsschema der Laufzeit (`en_US.UTF-8`).
- `PATH` – Der Ausführungspfad (`/usr/local/bin:/usr/bin/:/bin:/opt/bin`).
- `LD_LIBRARY_PATH` – Der Pfad der Systembibliothek (`/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib`).
- `NODE_PATH`— (Node.js) Der Node.js Bibliothekspfad (`/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules`).
- `NODE_OPTIONS`— (Node.js) Für Node.js Laufzeiten können Sie experimentelle Funktionen `NODE_OPTIONS` wieder aktivieren, die Lambda standardmäßig deaktiviert.
- `PYTHONPATH` – (Python) Der Python-Bibliothekspfad (`$LAMBDA_RUNTIME_DIR`).
- `GEM_PATH` – (Ruby) Der Pfad der Ruby-Bibliothek (`$LAMBDA_TASK_ROOT/vendor/bundle/ruby/3.3.0:/opt/ruby/gems/3.3.0`).
- `AWS_XRAY_CONTEXT_MISSING`— Für die X-Ray Ablaufverfolgung legt Lambda dies auf fest, um `LOG_ERROR` zu vermeiden, dass Laufzeitfehler vom X-Ray SDK ausgelöst werden.
- `AWS_XRAY_DAEMON_ADDRESS`— Für die X-Ray Ablaufverfolgung die IP-Adresse und den Port des Daemons. X-Ray
- `AWS_LAMBDA_DOTNET_PREJIT` – (.NET) Legen Sie für die .NET-Laufzeitumgebung diese Variable so fest, dass sie .NET-spezifische Laufzeitoptimierungen aktiviert oder deaktiviert. Zu den Werten gehören `always`, `never` und `provisioned-concurrency`. Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität für eine Event-Funktion.

- TZ – Die Zeitzone der Umgebung (`:UTC`). Die Ausführungsumgebung verwendet NTP, um die Systemuhr zu synchronisieren.

Die angezeigten Beispielwerte spiegeln die neuesten Laufzeiten wider. Das Vorhandensein bestimmter Variablen oder deren Werte kann je nach früheren Laufzeiten unterschiedlich sein.

Sicherung von Lambda-Umgebungsvariablen

Sie können Ihre Umgebungsvariablen sichern, können Sie eine serverseitige Verschlüsselung nutzen, um Ihre Data-at-Rest zu schützen und eine clientseitige Verschlüsselung, um Ihre Daten während der Übertragung zu schützen.

> **Note**
>
> Um die Datenbanksicherheit zu erhöhen, empfehlen wir, AWS Secrets Manager anstelle von Umgebungsvariablen zum Speichern von Datenbankanmeldeinformationen zu verwenden. Weitere Informationen finden Sie unter Verwenden Sie Secrets Manager Manager-Geheimnisse in Lambda-Ereignisfunktionen.

Sicherheit im Ruhezustand

Lambda bietet im Ruhezustand immer serverseitiger Verschlüsselung mit einem AWS KMS key an. Standardmäßig verwendet Lambda einen Von AWS verwalteter Schlüssel. Wenn dieses Standardverhalten Ihrem Workflow entspricht, müssen Sie nichts anderes einrichten. Lambda erstellt das Von AWS verwalteter Schlüssel in Ihrem Konto und verwaltet die Berechtigungen für Sie. AWS berechnet Ihnen keine Gebühren für die Verwendung dieses Schlüssels.

Wenn Sie möchten, können Sie stattdessen einen vom AWS KMS Kunden verwalteten Schlüssel bereitstellen. Sie können dies tun, um die Drehung des KMS-Schlüssels zu steuern oder die Anforderungen Ihrer Organisation für die Verwaltung von KMS-Schlüsseln zu erfüllen. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, können nur Benutzer in Ihrem Konto mit Zugriff auf den KMS-Schlüssel Umgebungsvariablen für die Funktion anzeigen oder verwalten.

Für vom Kunden verwaltete Schlüssel fallen AWS KMS Standardgebühren an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Sicherheit während der Übertragung

Für zusätzliche Sicherheit können Sie Helfer für die Verschlüsselung bei der Übertragung aktivieren, wodurch sichergestellt wird, dass Ihre Umgebungsvariablen clientseitig verschlüsselt sind, um sie während der Übertragung zu schützen.

So konfigurieren Sie die Verschlüsselung für Ihre Umgebungsvariablen

1. Verwenden Sie AWS Key Management Service (AWS KMS), um alle vom Kunden verwalteten Schlüssel für Lambda zu erstellen, die für die serverseitige und clientseitige Verschlüsselung verwendet werden sollen. Weitere Informationen finden Sie unter Erstellen von Schlüsseln im AWS Key Management Service -Entwicklerhandbuch.
2. Navigieren Sie mithilfe der Lambda-Konsole zur Seite Umgebungsvariablen bearbeiten.
   a. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
   b. Wählen Sie eine Funktion aus.
   c. Wählen Sie Konfiguration und dann Umgebungsvariablen aus der linken Navigationsleiste.
   d. Wählen Sie im Abschnitt Umgebungsvariablen Bearbeiten aus.
   e. Erweitern Sie Encryption configuration (Verschlüsselungskonfiguration).
3. (Optional) Aktivieren Sie Konsolenverschlüsselungshelfer, um die clientseitige Verschlüsselung zu verwenden, um Ihre Daten während der Übertragung zu schützen.
   a. Wählen Sie unter Verschlüsselung während der Übertragung die Option Helfer für die Verschlüsselung während der Übertragung aktivieren aus.
   b. Wählen Sie für jede Umgebungsvariable, für die Sie Konsolenverschlüsselungshilfen aktivieren möchten, neben der Umgebungsvariablen die Option Encrypt (Verschlüsseln) aus.
   c. Wählen AWS KMS key Sie unter Verschlüsselung bei der Übertragung einen vom Kunden verwalteten Schlüssel aus, den Sie zu Beginn dieses Verfahrens erstellt haben.
   d. Wählen Sie Ausführungsrollenrichtlinie und kopieren Sie die Richtlinie. Diese Richtlinie gewährt der Ausführungsrolle Ihrer Funktion die Berechtigung zum Entschlüsseln der Umgebungsvariablen.

      Speichern Sie diese Richtlinie zur Verwendung im letzten Schritt dieses Verfahrens.
   e. Fügen Sie Ihrer Funktion Code hinzu, der die Umgebungsvariablen entschlüsselt. Wählen Sie Geheimnisausschnitt entschlüsseln, um ein Beispiel anzuzeigen.
4. (Optional) Geben Sie Ihren kundenverwalteten Schlüssel zur Verschlüsselung im Ruhezustand an.

a. Wählen Sie Kundenmasterschlüssel verwenden aus.

b. Wählen Sie einen vom Kunden verwalteten Schlüssel aus, den Sie zu Beginn dieses Verfahrens erstellt haben.

5. Wählen Sie Speichern.

6. Richten Sie Berechtigungen ein.

   Wenn Sie einen vom Kunden verwalteten Schlüssel mit serverseitiger Verschlüsselung verwenden, erteilen Sie allen Benutzern oder Rollen Berechtigungen zum Anzeigen oder Verwalten von Umgebungsvariablen für die Funktion. Weitere Informationen finden Sie unter Verwalten von Berechtigungen für den serverseitigen Verschlüsselungs-KMS-Schlüssel.

   Wenn Sie die clientseitige Verschlüsselung für die Sicherheit während der Übertragung aktivieren, benötigt Ihre Funktion die Berechtigung zum Aufrufen der `kms:Decrypt`-API-Operation. Fügen Sie die Richtlinie, die Sie zuvor in diesem Verfahren gespeichert haben, zur Ausführungsrolle der Funktion hinzu.

Verwalten von Berechtigungen für den serverseitigen Verschlüsselungs-KMS-Schlüssel

Für Ihren Benutzer oder die Ausführungsrolle der Funktion sind keine AWS KMS Berechtigungen erforderlich, um den Standard-Verschlüsselungsschlüssel zu verwenden. Um einen vom kundenverwalteten Schlüssel verwenden zu können, benötigen Sie eine Berechtigungen zur Verwendung des Schlüssels. Lambda erstellt anhand Ihrer Berechtigungen eine Berechtigungserteilung für den Schlüssel. So wird es Lambda ermöglicht, diesen für die Verschlüsselung zu verwenden.

- `kms:ListAliases` – Zur Ansicht von Schlüssel in der Lambda-Konsole.
- `kms:CreateGrant`, `kms:Encrypt` – Zum Konfigurieren eines vom Kunden verwalteten Schlüssels für eine Funktion.
- `kms:Decrypt` – Zum Anzeigen und Verwalten von Umgebungsvariablen, die mit einem vom Kunden verwalteten Schlüssel verschlüsselt sind.

Sie können diese Berechtigungen über Ihre AWS-Konto oder über die ressourcenbasierte Berechtigungsrichtlinie eines Schlüssels erhalten. `ListAliases`wird durch die verwalteten Richtlinien für Lambda bereitgestellt. Schlüsselrichtlinien gewähren Benutzern in der Gruppe Key users (Schlüsselbenutzer) die verbleibenden Berechtigungen.

Benutzer ohne `Decrypt`-Berechtigungen können Funktionen weiterhin verwalten, aber sie können keine Umgebungsvariablen anzeigen oder in der Lambda-Konsole verwalten. Um zu verhindern, dass ein Benutzer Umgebungsvariablen anzeigen kann, fügen Sie den Berechtigungen des Benutzers eine Anweisung hinzu, die den Zugriff auf den Standardschlüssel, einen vom Kunden verwalteten Schlüssel oder alle Schlüssel verweigert.

Example IAM-Richtlinie – Verweigern des Zugriffs nach Schlüssel-ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-
xmpl-4be4-bc9d-0405a71945cc"
        }
    ]
}
```

Details zum Verwalten von Schlüsselberechtigungen finden Sie unter Key policies in AWS KMS im AWS Key Management Service -Entwicklerhandbuch.

Lambda Event Functions Zugriff auf Ressourcen in einer Amazon VPC gewähren

Mit Amazon Virtual Private Cloud (Amazon VPC) können Sie private Netzwerke in Ihren AWS-Konto Host-Ressourcen wie Amazon Elastic Compute Cloud (Amazon EC2) -Instances, Amazon Relational Database Service (Amazon RDS) -Instances und Amazon-Instances erstellen. ElastiCache Sie können Ihrer Lambda-Funktion Zugriff auf Ressourcen geben, die in einer Amazon VPC gehostet werden, indem Sie Ihre Funktion über die privaten Subnetze, die die Ressourcen enthalten, mit der VPC verbinden. Folgen Sie den Anweisungen in den folgenden Abschnitten, um eine Lambda-Funktion über die Lambda-Konsole, die AWS Command Line Interface (AWS CLI) oder an eine Amazon VPC anzuhängen. AWS SAM

> **Note**
>
> Jede Lambda-Funktion läuft in einer VPC, die dem Lambda-Dienst gehört und von diesem verwaltet wird. Diese VPCs werden automatisch von Lambda verwaltet und sind für Kunden nicht sichtbar. Die Konfiguration Ihrer Funktion für den Zugriff auf andere AWS Ressourcen in einer Amazon VPC hat keine Auswirkungen auf die Lambda-managed VPC, in der Ihre Funktion ausgeführt wird.

Sections

- Erforderliche IAM-Berechtigungen
- Hinzufügen von Lambda-Funktionen zu einer Amazon VPC in Ihrem AWS-Konto
- Internetzugang bei Verbindung mit einer VPC
- IPv6-Support
- Bewährte Methoden für die Verwendung von Lambda mit Amazon VPCs
- Hyperplane Elastic-Network-Schnittstelle (ENIs) verstehen
- Verwenden von IAM-Bedingungsschlüsseln für VPC-Einstellungen
- VPC-Tutorials

Erforderliche IAM-Berechtigungen

Um eine Lambda-Funktion an eine Amazon-VPC in Ihrer anzuhängen AWS-Konto, benötigt Lambda Berechtigungen zum Erstellen und Verwalten der Netzwerkschnittstellen, die es verwendet, um Ihrer Funktion Zugriff auf die Ressourcen in der VPC zu gewähren.

Die von Lambda erstellten Netzwerkschnittstellen werden als Hyperplane Elastic Network Interfaces oder Hyperplane ENIs bezeichnet. Weitere Informationen zu diesen Netzwerkschnittstellen finden Sie unter the section called “Hyperplane Elastic-Network-Schnittstelle (ENIs) verstehen”.

Sie können Ihrer Funktion die erforderlichen Berechtigungen erteilen, indem Sie die AWS verwaltete Richtlinie AWSLambdaVPCAccessExecutionRolean die Ausführungsrolle Ihrer Funktion anhängen. Wenn Sie eine neue Funktion in der Lambda-Konsole erstellen und sie an eine VPC anhängen, fügt Lambda diese Berechtigungsrichtlinie automatisch für Sie hinzu.

Wenn Sie es vorziehen, Ihre eigene IAM-Berechtigungsrichtlinie zu erstellen, stellen Sie sicher, dass Sie alle der folgenden Berechtigungen hinzufügen und sie auf allen Ressourcen erlauben (`"Resource": "*"`):

- ec2: CreateNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeSubnets
- ec2: DeleteNetworkInterface
- ec2: AssignPrivateIpAddresses
- ec2: UnassignPrivateIpAddresses

Beachten Sie, dass die Rolle Ihrer Funktion diese Berechtigungen nur benötigt, um die Netzwerkschnittstellen zu erstellen, nicht aber, um Ihre Funktion aufzurufen. Sie können Ihre Funktion immer noch erfolgreich aufrufen, wenn sie an eine Amazon VPC angehängt ist, auch wenn Sie diese Berechtigungen aus der Ausführungsrolle Ihrer Funktion entfernen.

Um Ihre Funktion an eine VPC anzuhängen, muss Lambda auch Netzwerkressourcen mithilfe Ihrer IAM-Benutzerrolle verifizieren. Stellen Sie sicher, dass Ihre Benutzerrolle über die folgenden IAM-Berechtigungen verfügt:

- ec2: DescribeSecurityGroups
- ec2: DescribeSubnets
- ec2: DescribeVpcs

- ec2: GetSecurityGroupsForVpc

> **Note**
>
> Die Amazon-EC2-Berechtigungen, die Sie der Ausführungsrolle Ihrer Funktion gewähren, werden vom Lambda-Service verwendet, um Ihre Funktion an eine VPC anzuhängen. Sie gewähren diese Berechtigungen jedoch auch implizit für den Code Ihrer Funktion. Das bedeutet, dass Ihr Funktionscode diese Amazon-EC2-API-Aufrufe ausführen kann. Ratschläge zur Einhaltung bewährter Sicherheitsverfahren finden Sie unter the section called “Bewährte Methoden für die Gewährleistung der Sicherheit”.

Hinzufügen von Lambda-Funktionen zu einer Amazon VPC in Ihrem AWS-Konto

Verbinden Sie Ihre Funktion mit einer Amazon-VPC in Ihrem, AWS-Konto indem Sie die Lambda-Konsole verwenden, oder. AWS CLI AWS SAM Wenn Sie das AWS CLI oder verwenden oder AWS SAM eine bestehende Funktion mithilfe der Lambda-Konsole an eine VPC anhängen, stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion über die erforderlichen Berechtigungen verfügt, die im vorherigen Abschnitt aufgeführt sind.

Lambda-Funktionen können keine direkte Verbindung mit einer VPC mit Dedicated-Instance-Tenancy herstellen. Zum Herstellen einer Verbindung mit Ressourcen in einer dedizierten VPC verbinden Sie sie mit einer zweiten VPC mit Standard-Tenancy.

Lambda console

So hängen Sie eine Funktion an eine Amazon VPC an, wenn Sie sie erstellen

1. Öffnen Sie die Seite „Event Functions“ der Lambda-Konsole und wählen Sie Create Event Function aus.
2. Geben Sie unter Basic information (Grundlegende Informationen) bei Function name (Funktionsname) einen Namen für Ihre Funktion ein.
3. Konfigurieren Sie die VPC-Einstellungen für die Funktion, indem Sie wie folgt vorgehen:
   a. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
   b. Wählen Sie VPC aktivieren und wählen Sie dann den VPC, dem Sie die Funktion zuordnen möchten.

c. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.

d. Wählen Sie die Teilnetze und Sicherheitsgruppen aus, für die die Netzwerkschnittstelle erstellt werden soll. Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

> **Note**
>
> Verbinden Sie Ihre Funktion für den Zugriff auf private Ressourcen mit privaten Subnetzen. Wenn Ihre Funktion Internetzugang benötigt, siehe the section called "Internetzugriff für VPC-Funktionen". Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie weder Internetzugang noch eine öffentliche IP-Adresse.

4. Wählen Sie „Event-Funktion erstellen".

So hängen Sie eine vorhandene Funktion an eine Amazon VPC an

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
3. Wählen Sie Bearbeiten aus.
4. Wählen Sie unter VPC die Amazon VPC aus, an die Sie Ihre Funktion anhängen möchten.
5. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
6. Wählen Sie die Teilnetze und Sicherheitsgruppen aus, für die die Netzwerkschnittstelle erstellt werden soll. Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

> **Note**
>
> Verbinden Sie Ihre Funktion für den Zugriff auf private Ressourcen mit privaten Subnetzen. Wenn Ihre Funktion Internetzugang benötigt, siehe the section called "Internetzugriff für VPC-Funktionen". Durch die Verbindung einer Funktion mit einem

öffentlichen Subnetz erhält sie weder Internetzugang noch eine öffentliche IP-Adresse.

7. Wählen Sie Speichern.

AWS CLI

So hängen Sie eine Funktion an eine Amazon VPC an, wenn Sie sie erstellen

- Um eine Lambda-Funktion zu erstellen und sie mit einer VPC zu verbinden, führen Sie den folgenden CLI `create-function`-Befehl aus.

```
aws lambda create-function --function-name my-function \
--runtime nodejs24.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

Geben Sie Ihre eigenen Subnetze und Sicherheitsgruppen an und setzen Sie `Ipv6AllowedForDualStack` je nach Anwendungsfall auf `true` oder `false`.

So hängen Sie eine vorhandene Funktion an eine Amazon VPC an

- Um eine vorhandene Funktion an eine VPC anzuhängen, führen Sie den folgenden CLI `update-function-configuration`-Befehl aus.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-0859123
```

So trennen Sie Ihre Funktion von einer VPC

- Um Ihre Funktion von einer VPC zu trennen, führen Sie den folgenden `update-function-configuration`-CLI-Befehl mit einer leeren Liste der VPC-Subnetze und Sicherheitsgruppen aus.

```
aws lambda update-function-configuration --function-name my-function \
```

```
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

So verbinden Sie Ihre Funktion mit einer VPC

- Um eine Lambda-Funktion an eine Amazon VPC anzuhängen, fügen Sie die `VpcConfig`-Eigenschaft zu Ihrer Funktionsdefinition hinzu, wie in der folgenden Beispielvorlage gezeigt. Weitere Informationen zu dieser Eigenschaft finden Sie unter AWS: :Lambda: :Function VpcConfig im CloudFormation Benutzerhandbuch (die AWS SAM `VpcConfig` Eigenschaft wird direkt an die `VpcConfig` Eigenschaft einer CloudFormation `AWS::Lambda::Function` Ressource übergeben).

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Security group for Lambda function
      VpcId: !Ref MyVPC

  MySubnet1:
    Type: AWS::EC2::Subnet
    Properties:
```

```
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.1.0/24

  MySubnet2:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.2.0/24

  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
```

Weitere Informationen zur Konfiguration Ihrer VPC in AWS SAM finden Sie unter AWS: :EC2: :VPC im Benutzerhandbuch.CloudFormation

Internetzugang bei Verbindung mit einer VPC

Standardmäßig haben Lambda-Funktionen Zugriff auf das öffentliche Internet. Wenn Sie Ihre Funktion an eine VPC anhängen, kann sie nur auf Ressourcen zugreifen, die in dieser VPC verfügbar sind. Um Ihrer Funktion Zugriff auf das Internet zu gewähren, müssen Sie die VPC auch für den Internetzugang konfigurieren. Weitere Informationen hierzu finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

IPv6-Support

Ihre Funktion kann über IPv6 eine Verbindung zu Ressourcen in Dual-Stack-VPC-Subnetzen herstellen. Diese Option ist standardmäßig deaktiviert. Um ausgehenden IPv6-Verkehr zuzulassen, verwenden Sie die Konsole oder die `--vpc-config Ipv6AllowedForDualStack=true`-Option mit dem Befehl create-function oder update-function-configuration.

> **Note**
>
> Um ausgehenden IPv6-Verkehr in einer VPC zuzulassen, müssen alle Subnetze, die mit der Funktion verbunden sind, Dual-Stack-Subnetze sein. Lambda unterstützt keine ausgehenden IPv6-Verbindungen für IPv6-only Subnetze in einer VPC oder ausgehende IPv6-Verbindungen für Funktionen, die nicht mit einer VPC verbunden sind.

Sie können Ihren Funktionscode aktualisieren, um explizit eine Verbindung zu Subnetzressourcen über IPv6 herzustellen. Das folgende Python-Beispiel öffnet einen Socket und stellt eine Verbindung zu einem IPv6-Server her.

Example— Verbindung zum IPv6-Server herstellen

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()
```

Bewährte Methoden für die Verwendung von Lambda mit Amazon VPCs

Um sicherzustellen, dass Ihre Lambda-VPC-Konfiguration den Best-Practice-Richtlinien entspricht, befolgen Sie die Hinweise in den folgenden Abschnitten.

Bewährte Methoden für die Gewährleistung der Sicherheit

Um Ihre Lambda-Funktion an eine VPC anzuhängen, müssen Sie der Ausführungsrolle Ihrer Funktion eine Reihe von Amazon EC2 EC2-Berechtigungen erteilen. Diese Berechtigungen sind erforderlich, um die Netzwerkschnittstellen zu erstellen, die Ihre Funktion für den Zugriff auf die Ressourcen in der VPC verwendet. Diese Berechtigungen werden jedoch auch implizit für den Code Ihrer Funktion

gewährt. Das bedeutet, dass Ihr Funktionscode die Erlaubnis hat, diese Amazon-EC2-API-Aufrufe zu tätigen.

Um dem Prinzip des Zugriffs mit den geringsten Rechten zu folgen, fügen Sie der Ausführungsrolle Ihrer Funktion eine Ablehnungsrichtlinie wie im folgenden Beispiel hinzu. Diese Richtlinie verhindert, dass Ihr Funktionscode Aufrufe an die Amazon-EC2-APIs sendet, während der Lambda-Dienst weiterhin VPC-Ressourcen in Ihrem Namen verwalten kann. Die Richtlinie verwendet den `lambda:SourceFunctionArn`-Bedingungsschlüssel, der nur für API-Aufrufe gilt, die während der Ausführung von Ihrem Funktionscode getätigt werden. Weitere Informationen finden Sie unter the section called “ARN der Quellfunktion”.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                 "ec2:CreateNetworkInterface",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeSubnets",
                 "ec2:DetachNetworkInterface",
                 "ec2:AssignPrivateIpAddresses",
                 "ec2:UnassignPrivateIpAddresses"
            ],
            "Resource": [ "*" ],
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-
west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS bietet Sicherheitsgruppen und Netzwerk-Zugriffskontrolllisten (ACLs), um die Sicherheit in Ihrer VPC zu erhöhen. Sicherheitsgruppen kontrollieren eingehenden und ausgehenden Verkehr für Ihre Ressourcen, Netzwerk-ACLs kontrollieren eingehenden und ausgehenden Zugriff für Ihre Subnetze. Sicherheitsgruppen bieten ausreichend Zugriffskontrolle für die meisten Subnetze. Sie können Netzwerk-ACLs verwenden, wenn Sie eine zusätzliche Sicherheitsebene für Ihre VPC benötigen. Allgemeine Richtlinien zu bewährten Sicherheitspraktiken bei der Verwendung von Amazon VPCs finden Sie unter Bewährte Sicherheitspraktiken für Ihre VPC im Amazon-Virtual-Private Cloud-Benutzerhandbuch.

Bewährte Methoden zur Leistungssteigerung

Wenn Sie Ihre Funktion an eine VPC anhängen, prüft Lambda, ob es eine verfügbare Netzwerkressource (Hyperplane ENI) gibt, mit der es eine Verbindung herstellen kann. Hyperplane-ENIs sind einer bestimmten Kombination von Sicherheitsgruppen und VPC-Subnetzen zugeordnet. Wenn Sie bereits eine Funktion an eine VPC angehängt haben, bedeutet die Angabe derselben Subnetze und Sicherheitsgruppen beim Anhängen einer anderen Funktion, dass Lambda die Netzwerkressourcen gemeinsam nutzen kann und die Notwendigkeit entfällt, ein neues Hyperplane-ENI zu erstellen. Weitere Informationen zu Hyperplane-ENIs und ihrem Lebenszyklus finden Sie unter the section called “Hyperplane Elastic-Network-Schnittstelle (ENIs) verstehen”.

Hyperplane Elastic-Network-Schnittstelle (ENIs) verstehen

Eine Hyperplane ENI ist eine verwaltete Ressource, die als Netzwerkschnittstelle zwischen Ihrer Lambda-Funktion und den Ressourcen fungiert, mit denen Sie Ihre Funktion verbinden möchten. Der Lambda-Service erstellt und verwaltet diese ENIs automatisch, wenn Sie Ihre Funktion an eine VPC anhängen.

Hyperplane-ENIs sind für Sie nicht direkt sichtbar, und Sie müssen sie nicht konfigurieren oder verwalten. Wenn Sie jedoch wissen, wie sie funktionieren, können Sie besser verstehen, wie sich Ihre Funktion verhält, wenn Sie sie an eine VPC anhängen.

Wenn Sie zum ersten Mal eine Funktion mithilfe einer bestimmten Kombination aus Subnetz und Sicherheitsgruppe an eine VPC anfügen, erstellt Lambda eine Hyperplane-ENI. Andere Funktionen in Ihrem Konto, die dieselbe Kombination aus Subnetz und Sicherheitsgruppe verwenden, können diese ENI ebenfalls verwenden. Wo immer möglich, verwendet Lambda bestehende ENIs wieder, um die Ressourcennutzung zu optimieren und die Erstellung neuer ENIs zu minimieren. Jedes Hyperplane ENI unterstützt bis zu 65.000. connections/ports Wenn die Anzahl der Verbindungen diese Grenze überschreitet, skaliert Lambda die Anzahl der ENIs automatisch auf der Grundlage des Netzwerkverkehrs und der Gleichzeitigkeitsanforderungen.

Bei neuen Funktionen bleibt Ihre Funktion, während Lambda eine Hyperplane-ENI erstellt, im Status Ausstehend und Sie können sie nicht aufrufen. Ihre Funktion wechselt erst in den Status Aktiv, wenn die Hyperplane ENI bereit ist, was mehrere Minuten dauern kann. Für bestehende Funktionen können Sie keine zusätzlichen Operationen ausführen, die auf die Funktion abzielen, wie z. B. das Erstellen von Versionen oder das Aktualisieren des Funktionscodes, aber Sie können weiterhin frühere Versionen der Funktion aufrufen.

Im Rahmen der Verwaltung des ENI-Lebenszyklus kann Lambda ENIs löschen und neu erstellen, um den Netzwerkverkehr zwischen ENIs auszugleichen oder um Probleme zu beheben, die bei ENI-Integritätsprüfungen festgestellt wurden. Wenn eine Lambda-Funktion 14 Tage lang inaktiv bleibt, fordert Lambda außerdem alle ungenutzten Hyperplane-ENIs zurück und setzt den Funktionsstatus auf `Inactive`. Der nächste Aufrufversuch schlägt fehl, und die Funktion geht wieder in den Zustand „Ausstehend“ über, bis Lambda die Erstellung oder Zuweisung einer Hyperplane-ENI abgeschlossen hat. Wir empfehlen, dass Ihr Design nicht auf der Persistenz von ENIs basiert.

Wenn Sie eine Funktion aktualisieren, um ihre VPC-Konfiguration zu entfernen, benötigt Lambda bis zu 20 Minuten, um die angehängte Hyperplane-ENI zu löschen. Lambda löscht die ENI nur, wenn keine andere Funktion (oder veröffentlichte Funktionsversion) diese Hyperplane ENI verwendet.

Lambda verwendet Berechtigungen in der Ausführungsrolle der Funktion, um die Hyperplane ENI zu löschen. Wenn Sie die Ausführungsrolle löschen, bevor Lambda die Hyperplane ENI löscht, kann Lambda die Hyperplane ENI nicht löschen. Sie können den Löschvorgang manuell durchführen.

Verwenden von IAM-Bedingungsschlüsseln für VPC-Einstellungen

Sie können Lambda-specific Bedingungsschlüssel für VPC-Einstellungen verwenden, um zusätzliche Berechtigungskontrollen für Ihre Lambda-Funktionen bereitzustellen. Sie können beispielsweise festlegen, dass alle Funktionen in Ihrer Organisation mit einer VPC sein müssen. Sie können auch die Subnetze und Sicherheitsgruppen angeben, die die Benutzer der Funktion verwenden können bzw. nicht verwenden können.

Lambda unterstützt die folgenden IAM-Bedingungsschlüssel:

- lambda: VpcIds — Erlauben oder verweigern Sie eine oder mehrere VPCs.
- lambda: SubnetIds — Erlaubt oder verweigert ein oder mehrere Subnetze.
- lambda: SecurityGroupIds — Erlaubt oder verweigert eine oder mehrere Sicherheitsgruppen.

Die Lambda-API funktioniert CreateFunctionund UpdateFunctionConfigurationunterstützt diese Bedingungsschlüssel. Weitere Informationen zur Verwendung von Bedingungsschlüsseln in IAM-Richtlinien finden Sie unter IAM-JSON-Richtlinienelemente: Bedingung im IAM-Benutzerhandbuch.

> **Tip**
>
> Wenn Ihre Funktion bereits eine VPC-Konfiguration aus einer früheren API-Anforderung enthält, können Sie eine `UpdateFunctionConfiguration`-Anforderung ohne die VPC-Konfiguration senden.

Beispielrichtlinien mit Bedingungsschlüsseln für VPC-Einstellungen

In den folgenden Beispielen wird gezeigt, wie Bedingungsschlüssel für VPC-Einstellungen verwendet werden. Nachdem Sie eine Richtlinienanweisung mit den gewünschten Einschränkungen erstellt haben, fügen Sie die Richtlinienanweisung für den -Zielbenutzer oder die Zielrolle an.

Stellen Sie sicher, dass Benutzer nur VPC-connected Funktionen bereitstellen

Um sicherzustellen, dass alle Benutzer nur VPC-connected Funktionen bereitstellen, können Sie Vorgänge zum Erstellen und Aktualisieren von Funktionen verweigern, die keine gültige VPC-ID enthalten.

Beachten Sie, dass die VPC-ID kein Eingabeparameter für die `CreateFunction`- oder `UpdateFunctionConfiguration` -Anforderung ist. Lambda ruft den VPC-ID-Wert basierend auf den Subnetz- und Sicherheitsgruppenparametern ab.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
```

```
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

Benutzern den Zugriff auf bestimmte VPCs, Subnetze oder Sicherheitsgruppen verweigern

Um Benutzern den Zugriff auf bestimmte VPCs `StringEquals` zu verweigern, überprüfen Sie den Wert der `lambda:VpcIds`-Bedingung. Im folgenden Beispiel wird Benutzern der Zugriff auf `vpc-1` und `vpc-2` verweigert.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceOutOfVPC",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

Um Benutzern den Zugriff auf bestimmte Subnetze zu verweigern, verwenden Sie `StringEquals`, um den Wert der `lambda:SubnetIds` Bedingung zu überprüfen. Im folgenden Beispiel wird Benutzern der Zugriff auf `subnet-1` und `subnet-2` verweigert.

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

Um Benutzern den Zugriff auf bestimmte Sicherheitsgruppen zu verweigern, verwenden Sie `StringEquals`, um den Wert der `lambda:SecurityGroupIds`-Bedingung zu überprüfen. Im folgenden Beispiel wird Benutzern der Zugriff auf `sg-1` und `sg-2` verweigert.

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

Benutzern das Erstellen und Aktualisieren von Funktionen mit bestimmten VPC-Einstellungen ermöglichen

Um Benutzern den Zugriff auf bestimmte VPCs zu ermöglichen, verwenden Sie `StringEquals`, um den Wert der `lambda:VpcIds`-Bedingung zu überprüfen. Im folgenden Beispiel können Benutzer auf `vpc-1` und `vpc-2` zugreifen.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceStayInSpecificVpc",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

Um Benutzern den Zugriff auf bestimmte Subnetze zu ermöglichen, verwenden Sie `StringEquals`, um den Wert der `lambda:SubnetIds`-Bedingung zu überprüfen. Im folgenden Beispiel können Benutzer auf `subnet-1` und `subnet-2` zugreifen.

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
```

```
            "lambda:UpdateFunctionConfiguration"
        ],
        "Effect": "Allow",
        "Resource": "*",
        "Condition": {
          "ForAllValues:StringEquals": {
              "lambda:SubnetIds": ["subnet-1", "subnet-2"]
          }
        }
    }
```

Um Benutzern den Zugriff auf bestimmte Sicherheitsgruppen zu ermöglichen, verwenden Sie `StringEquals`, um den Wert der `lambda:SecurityGroupIds`-Bedingung zu überprüfen. Im folgenden Beispiel können Benutzer auf `sg-1` und `sg-2` zugreifen.

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC-Tutorials

In den folgenden Tutorials verbinden Sie eine Lambda-Funktion mit Ressourcen in Ihrer VPC.

- Tutorial: Verwenden einer Lambda-Funktion für den Zugriff auf Amazon RDS in einer Amazon VPC
- Tutorial: Konfiguration einer Lambda-Funktion für den Zugriff auf Amazon ElastiCache in einer Amazon VPC

Informationen zum Verbinden Ihrer Funktion mit Ressourcen in einer VPC, die zu einer anderen gehört AWS-Konto, finden Sie unterthe section called “Funktionen an Ressourcen in einem anderen Konto anhängen”.

Lambda Event Functions Zugriff auf eine Ressource in einer Amazon VPC in einem anderen Konto gewähren

Sie können Ihrer AWS Lambda Funktion Zugriff auf eine Ressource in einer Amazon VPC in Amazon Virtual Private Cloud gewähren, die von einem anderen Konto verwaltet wird, ohne dass eine der VPC dem Internet ausgesetzt wird. Mit diesem Zugriffsmuster können Sie Daten mit anderen Organisationen gemeinsam nutzen. AWS Mithilfe dieses Zugriffsmusters können Sie Daten zwischen VPCs mit einem höheren Sicherheits- und Leistungsniveau austauschen als über das Internet. Konfigurieren Sie Ihre Lambda-Funktion so, dass sie eine Amazon VPC-Peering-Verbindung für den Zugriff auf diese Ressourcen verwendet.

> **Warning**
>
> Wenn Sie den Zugriff zwischen Konten oder VPCs zulassen, überprüfen Sie, ob Ihr Plan die Sicherheitsanforderungen der jeweiligen Organisationen erfüllt, die diese Konten verwalten. Wenn Sie die Anweisungen in diesem Dokument befolgen, wirkt sich dies auf den Sicherheitsstatus Ihrer Ressourcen aus.

In diesem Tutorial verbinden Sie zwei Konten über eine Peering-Verbindung mithilfe von IPv4. Sie konfigurieren eine Lambda-Funktion, die noch nicht mit einer Amazon VPC verbunden ist. Sie konfigurieren die DNS-Auflösung, um Ihre Funktion mit Ressourcen zu verbinden, die keine statischen IPs bereitstellen. Informationen zur Anpassung dieser Anweisungen an andere Peering-Szenarien finden Sie im VPC Peering Guide.

Voraussetzungen

Um einer Lambda-Funktion Zugriff auf eine Ressource in einem anderen Konto zu gewähren, benötigen Sie:

- Eine Lambda-Funktion, die so konfiguriert ist, dass sie sich mit Ihrer Ressource authentifiziert und dann aus ihr liest.
- Eine Ressource in einem anderen Konto, z. B. einem Amazon-RDS-Cluster, ist über Amazon VPC verfügbar.
- Anmeldeinformationen für das Konto Ihrer Lambda-Funktion und das Konto Ihrer Ressource. Wenn Sie nicht berechtigt sind, das Konto Ihrer Ressource zu verwenden, wenden Sie sich an einen autorisierten Benutzer, um dieses Konto einzurichten.

- Berechtigung zum Erstellen und Aktualisieren einer VPC (und zur Unterstützung von Amazon-VPC-Ressourcen), die mit Ihrer Lambda-Funktion verknüpft werden soll.
- Berechtigung zum Aktualisieren der Ausführungsrolle und der VPC-Konfiguration für Ihre Lambda-Funktion.
- Berechtigung zum Erstellen einer VPC-Peering-Verbindung im Konto Ihrer Lambda-Funktion.
- Berechtigung zum Annehmen einer VPC-Peering-Verbindung im Konto Ihrer Ressource.
- Erlaubnis, die Konfiguration der VPC Ihrer Ressource (und der Unterstützung von Amazon-VPC-Ressourcen) zu aktualisieren.
- Berechtigung zum Aufrufen Ihrer Lambda-Funktion.

Erstellen Sie eine Amazon VPC im Konto Ihrer Funktion

Erstellen Sie eine Amazon VPC, Subnetze, Routing-Tabellen und eine Sicherheitsgruppe im Konto Ihrer Lambda-Funktion.

So erstellen Sie eine VPC, Subnetze und weitere VPC-Ressourcen mit der Konsole

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie auf dem VPC-Dashboard VPC erstellen aus.
3. Geben Sie für den IPv4-CIDR-Block einen privaten CIDR-Block an. Ihr CIDR-Block darf sich nicht mit Blöcken überlappen, die in der VPC Ihrer Ressource verwendet werden. Wählen Sie keinen Block, den Ihre Ressourcen-VPC verwendet, um den Ressourcen IPs zuzuweisen, oder einen Block, der bereits in den Routentabellen in Ihrer Ressourcen-VPC definiert ist. Weitere Informationen zur Definition geeigneter CIDR-Blöcke finden Sie unter VPC CIDR-Blöcke.
4. Wählen Sie AZs anpassen aus.
5. Wählen Sie dieselben AZs wie Ihre Ressource aus.
6. Wählen Sie für Number of public subnets (Anzahl der öffentlichen Subnetze) 0 aus.
7. Wählen Sie für VPC endpoints (VPC-Endpunkte) None (Keine) aus.
8. Wählen Sie VPC erstellen aus.

Gewährung von VPC-Berechtigungen für die Ausführungsrolle Ihrer Funktion

Stellen Sie eine Verbindung AWSLambdaVPCAccessExecutionRolezur Ausführungsrolle Ihrer Funktion her, damit diese eine Verbindung zu VPCs herstellen kann.

So gewähren Sie der Ausführungsrolle Ihrer Funktion VPC-Berechtigungen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie Konfiguration.
4. Wählen Sie Berechtigungen.
5. Wählen Sie unter Rollenname die Ausführungsrolle.
6. Wählen Sie im Abschnitt Berechtigungsrichtlinien die Option Berechtigungen hinzufügen.
7. Wählen Sie in der Dropdown-Liste Richtlinien anhängen.
8. Geben Sie in das Suchfeld `AWSLambdaVPCAccessExecutionRole` ein.
9. Aktivieren Sie links neben dem Richtliniennamen das Kontrollkästchen.
10. Wählen Sie Add permissions (Berechtigungen hinzufügen) aus.

Um Ihre Funktion an Ihre Amazon VPC anzuhängen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
4. Wählen Sie Bearbeiten aus.
5. Wählen Sie unter VPC Ihre VPC aus
6. Wählen Sie unter Subnetze Ihre Subnetze aus.
7. Wählen Sie unter Sicherheitsgruppen die Standardsicherheitsgruppe für Ihre VPC.
8. Wählen Sie Speichern.

Erstellen Sie eine VPC-Peering-Verbindungsanfrage

Erstellen Sie eine VPC-Peering-Verbindungsanforderung von der VPC Ihrer Funktion (der VPC des Anforderers) zur VPC Ihrer Ressource (der Akzepter-VPC).

So erstellen Sie eine VPC-Peering-Verbindung von der VPC Ihrer Funktion

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie Create peering connection (Peering-Verbindung erstellen).

4. Wählen Sie für VPC-ID (Requester) die VPC Ihrer Funktion aus.
5. Geben Sie unter Konto-ID die ID des Kontos Ihrer Ressource ein.
6. Geben Sie für VPC ID (Accepter) die VPC Ihrer Ressource ein.

Bereiten Sie das Konto Ihrer Ressource vor

Um Ihre Peering-Verbindung herzustellen und die VPC Ihrer Ressource für die Verwendung der Verbindung vorzubereiten, melden Sie sich beim Konto Ihrer Ressource mit einer Rolle an, die die in den Voraussetzungen aufgeführten Berechtigungen besitzt. Die Schritte zur Anmeldung können je nachdem, wie das Konto gesichert ist, unterschiedlich sein. Weitere Informationen zur Anmeldung bei einem AWS Konto finden Sie im AWS Sign-in Benutzerhandbuch. Führen Sie im Konto Ihrer Ressource die folgenden Verfahren aus.

So akzeptieren Sie die VPC-Peering-Verbindungsanfrage

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie erst die ausstehende VPC-Peering-Verbindung (der Status lautet pending-acceptance) aus.
4. Wählen Sie Aktionen.
5. Wählen Sie in der Dropdown-Liste die Option Anfrage annehmen aus.
6. Wenn Sie zur Bestätigung aufgefordert werden, wählen Sie Akzeptieren aus.
7. Wählen Sie Meine Routing-Tabellen jetzt ändern, um der Haupt-Routing-Tabelle für Ihre VPC eine Route hinzuzufügen, sodass Sie Traffic über die Peering-Verbindung senden und empfangen können.

Untersuchen Sie die Routentabellen für die VPC der Ressource. Die von Amazon VPC generierte Route stellt möglicherweise keine Konnektivität her, je nachdem, wie die VPC Ihrer Ressource eingerichtet ist. Suchen Sie nach Konflikten zwischen der neuen Route und der vorhandenen Konfiguration für die VPC. Weitere Informationen zur Fehlerbehebung finden Sie unter Problembehandlung bei einer VPC-Peering-Verbindung im Amazon Virtual Private Cloud VPC Peering Guide.

So aktualisieren Sie die Sicherheitsgruppe für Ihre Ressource

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.

2. Wählen Sie im Navigationsbereich Sicherheitsgruppen aus.
3. Wählen Sie die Sicherheitsgruppe für Ihre Ressource aus.
4. Wählen Sie Aktionen.
5. Wählen Sie in der Dropdownliste die Option Regeln für eingehenden Datenverkehr bearbeiten aus.
6. Wählen Sie Regel hinzufügen aus.
7. Geben Sie als Quelle die Konto-ID und die Sicherheitsgruppen-ID Ihrer Funktion ein, getrennt durch einen Schrägstrich (z. B. 111122223333/sg -1a2b3c4d).
8. Wählen Sie Edit outbound rules (Regeln für ausgehenden Datenverkehr bearbeiten).
9. Prüfen Sie, ob der ausgehende Verkehr eingeschränkt ist. Die VPC-Standardeinstellungen gestatten allen ausgehenden Datenverkehr. Wenn der ausgehende Datenverkehr eingeschränkt ist, fahren Sie mit dem nächsten Schritt fort.
10. Wählen Sie Regel hinzufügen aus.
11. Geben Sie als Ziel die Konto-ID und die Sicherheitsgruppen-ID Ihrer Funktion ein, getrennt durch einen Schrägstrich (z. B. -1a2b3c4d). 111122223333/sg
12. Wählen Sie Regeln speichern aus.

Aktivieren einer DNS-Auflösung für Ihre Peering-Verbindung

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie Ihre Peering-Verbindung aus.
4. Wählen Sie Aktionen.
5. Wählen Sie DNS-Einstellungen ändern aus.
6. Wählen Sie unter Akzepter-DNS-Auflösung die Option Erlauben Sie dem anfragenden VPC, DNS von akzeptierenden VPC-Hosts in private IP aufzulösen aus.
7. Wählen Sie Änderungen speichern aus.

Aktualisieren Sie die VPC-Konfiguration im Konto Ihrer Funktion

Melden Sie sich beim Konto Ihrer Funktion an und aktualisieren Sie die VPC-Konfiguration.

So fügen Sie Ihrer VPC-Peering-Verbindung eine Route hinzu

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Route Tables (Routing-Tabellen) aus.
3. Aktivieren Sie das Kontrollkästchen neben dem Namen der Routentabelle für das Teilnetz, das Sie mit Ihrer Funktion verknüpft haben.
4. Wählen Sie Aktionen.
5. Wählen Sie Routen bearbeiten aus.
6. Wählen Sie Route hinzufügen aus.
7. Geben Sie unter Ziel den CIDR-Block für die VPC Ihrer Ressource ein.
8. Wählen Sie als Ziel Ihre VPC-Peering-Verbindung aus.
9. Wählen Sie Änderungen speichern aus.

Weitere Informationen zu Überlegungen, auf die Sie bei der Aktualisierung Ihrer Routentabellen möglicherweise stoßen, finden Sie unter Aktualisieren Sie Ihre Routentabellen für eine VPC-Peering-Verbindung.

Um die Sicherheitsgruppe für Ihre Lambda-Funktion zu aktualisieren

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Sicherheitsgruppen aus.
3. Wählen Sie Aktionen.
4. Wählen Sie Edit inbound rules (Regeln für eingehenden Datenverkehr bearbeiten) aus.
5. Wählen Sie Regel hinzufügen aus.
6. Geben Sie als Quelle die Konto-ID und die Sicherheitsgruppen-ID Ihrer Ressource ein, getrennt durch einen Schrägstrich (z. B. 111122223333/sg -1a2b3c4d).
7. Wählen Sie Regeln speichern aus.

Aktivieren einer DNS-Auflösung für Ihre Peering-Verbindung

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie Ihre Peering-Verbindung aus.
4. Wählen Sie Aktionen.

5. Wählen Sie DNS-Einstellungen ändern aus.
6. Wählen Sie unter DNS-Auflösung des Antragstellers die Option Zulassen, dass der akzeptierende VPC die DNS von VPC-Hosts des Antragstellers in eine private IP auflöst aus.
7. Wählen Sie Änderungen speichern aus.

Testen der -Funktion

So erstellen Sie ein Testereignis und überprüfen die Ausgabe Ihrer Funktion

1. Wählen Sie im Bereich Codequelle die Option Testen aus.
2. Wählen Sie Neues Ereignis erstellen.
3. Ersetzen Sie im Bereich Event JSON die Standardwerte durch eine Eingabe, die für Ihre Lambda-Funktion geeignet ist.
4. Wählen Sie aufrufen aus.
5. Vergewissern Sie sich auf der Registerkarte Ausführungsergebnisse, dass die Antwort Ihre erwartete Ausgabe enthält.

Darüber hinaus können Sie die Protokolle Ihrer Funktion überprüfen, um sicherzustellen, dass die Protokolle Ihren Erwartungen entsprechen.

Um die Aufrufaufzeichnungen Ihrer Funktion in Logs einzusehen CloudWatch

1. Wählen Sie den Tab Überwachung.
2. Wählen Sie CloudWatch Protokolle anzeigen aus.
3. Wählen Sie auf der Registerkarte Protokollstreams den Protokollstream für den Aufruf Ihrer Funktion aus.
4. Vergewissern Sie sich, dass Ihre Protokolle Ihren Erwartungen entsprechen.

Aktivieren Sie den Internetzugang für VPC-connected Lambda Event Functions

Standardmäßig werden Lambda-Funktionen in einer Lambda-managed VPC mit Internetzugang ausgeführt. Um auf Ressourcen in einer VPC in Ihrem Konto zuzugreifen, können Sie einer Funktion eine VPC-Konfiguration hinzufügen. Dadurch wird die Funktion auf Ressourcen innerhalb dieser VPC beschränkt, es sei denn, die VPC hat Internetzugang. Auf dieser Seite wird erklärt, wie Sie den Internetzugang für VPC-connected Lambda-Funktionen bereitstellen.

Ich habe noch keine VPC

Erstellen Sie die VPC

Der VPC-Workflow erstellen erstellt alle VPC-Ressourcen, die für eine Lambda-Funktion für den Zugriff auf das öffentliche Internet von einem privaten Subnetz aus erforderlich sind, einschließlich Subnetzen, NAT-Gateways, Internet-Gateways und Routing-Tabelleneinträgen.

So erstellen Sie die VPC

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie auf dem VPC-Dashboard VPC erstellen aus.
3. Wählen Sie unter Zu erstellende Ressourcen die Option VPC und mehr aus.
4. Konfigurieren Sie die VPC
    a. Geben Sie unter Name tag auto-generation (Automatische Generierung des Namens-Tags) einen Namen für die VPC ein.
    b. Behalten Sie für den IPv4-CIDR-Block entweder den Standardvorschlag bei oder geben Sie den für Ihre Anwendung oder Ihr Netzwerk erforderlichen CIDR-Block ein.
    c. Wenn Ihre Anwendung über IPv6-Adressen kommuniziert, wählen Sie IPv6 CIDR-Block, IPv6 CIDR-Block. Amazon-provided
5. Konfiguration der Subnetze
    a. Wählen Sie für Anzahl der Availability Zones 2 aus. Wir empfehlen mindestens zwei AZs für hohe Verfügbarkeit.
    b. Wählen Sie für Number of public subnets (Anzahl der öffentlichen Subnetze) 2 aus.
    c. Wählen Sie für Number of private subnets (Anzahl der privaten Subnetze) 2 aus.

d. Sie können die standardmäßigen CIDR-Blöcke für die Subnetze beibehalten oder alternativ CIDR-Blöcke des Subnetzes anpassen erweitern und einen CIDR-Block eingeben. Weitere Informationen finden Sie unter CIDR-Blöcke für das Subnetz.

6. Wählen Sie für NAT-Gateways 1 pro AZ aus, um die Resilienz zu verbessern.
7. Wählen Sie für ein Internet-Gateway nur für ausgehenden Datenverkehr die Option Ja aus, wenn Sie sich dafür entschieden haben, einen IPv6-CIDR-Block einzubeziehen.
8. Behalten Sie für VPC-Endpunkte den Standardwert (S3-Gateway) bei. Für diese Option fallen keine Kosten an. Weitere Informationen finden Sie unter Arten von VPC-Endpunkten für Amazon S3.
9. Behalten Sie für die DNS-Optionen die Standardeinstellungen bei.
10. Wählen Sie VPC erstellen aus.

Konfigurieren der Lambda-Funktion

So konfigurieren Sie eine VPC bei der Funktionserstellung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Geben Sie unter Basic information (Grundlegende Informationen) bei Function name (Funktionsname) einen Namen für Ihre Funktion ein.
4. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
5. Wählen Sie VPC aktivieren und wählen Sie dann eine VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

> **Note**
>
> Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie „Event-Funktion erstellen“.

Lambda erstellt automatisch eine Ausführungsrolle mit der AWSLambdaVPCAccessExecutionRole AWS verwalteten Richtlinie. Die Berechtigungen in dieser Richtlinie sind nur zum Erstellen von Elastic Network-Schnittstellen für die VPC-Konfiguration und nicht zum Aufrufen Ihrer Funktion erforderlich. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die AWSLambdaVPCAccessExecutionRoleRichtlinie aus Ihrer Ausführungsrolle entfernen, nachdem Sie die Funktion und die VPC-Konfiguration erstellt haben. Weitere Informationen finden Sie unter Erforderliche IAM-Berechtigungen.

So konfigurieren Sie eine VPC für eine vorhandene Funktion

Um einer vorhandenen Funktion eine VPC-Konfiguration hinzuzufügen, muss die Ausführungsrolle der Funktion über die Berechtigung verfügen, elastische Netzwerkschnittstellen zu erstellen und zu verwalten. Die AWSLambdaVPCAccessExecutionRole AWS verwaltete Richtlinie umfasst die erforderlichen Berechtigungen. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die AWSLambdaVPCAccessExecutionRoleRichtlinie aus Ihrer Ausführungsrolle entfernen, nachdem Sie die VPC-Konfiguration erstellt haben.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
4. Wählen Sie unter VPC die Option Edit (Bearbeiten) aus.
5. Wählen Sie die VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

> **Note**
>
> Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

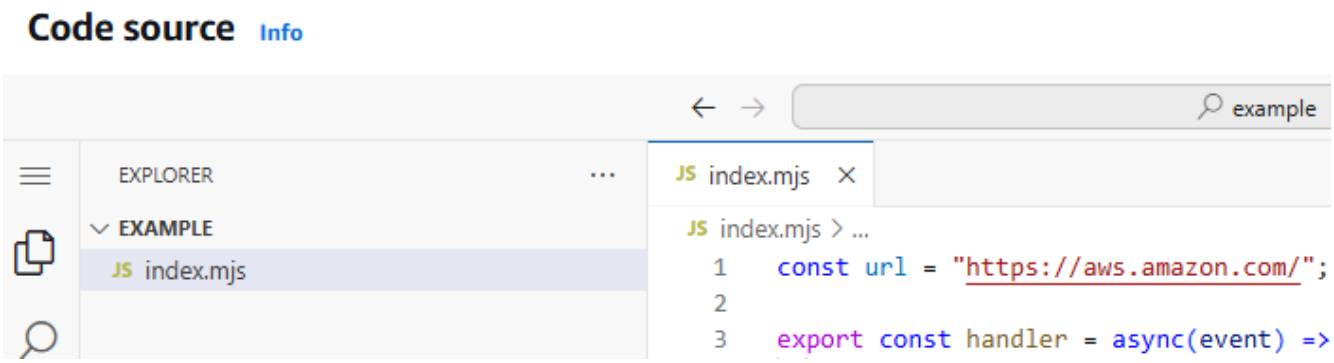
8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie Speichern.

Testen der Funktion

Verwenden Sie den folgenden Beispielcode, um zu bestätigen, dass Ihre VPC-connected Funktion das öffentliche Internet erreichen kann. Bei Erfolg gibt der Code einen `200`-Statuscode zurück. Ist dies nicht der Fall, wird die Funktion abgebrochen.

Node.js

1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei index.mjs ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.

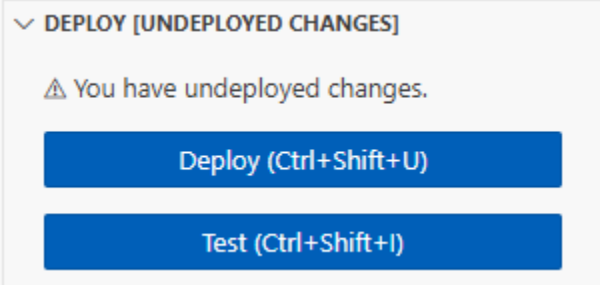


Example— HTTP-Anfrage mit async/await

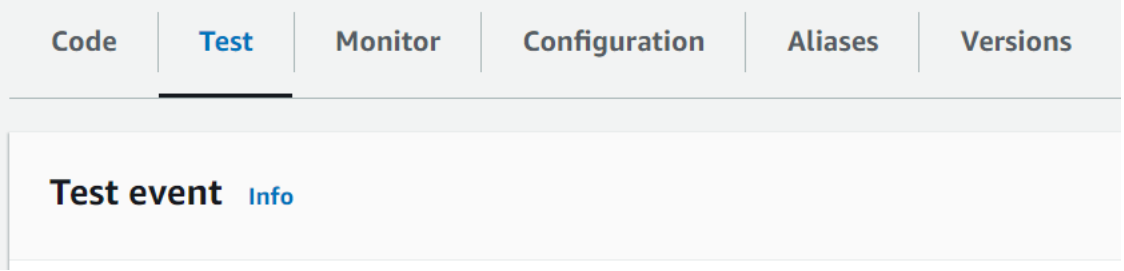
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
```

```
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

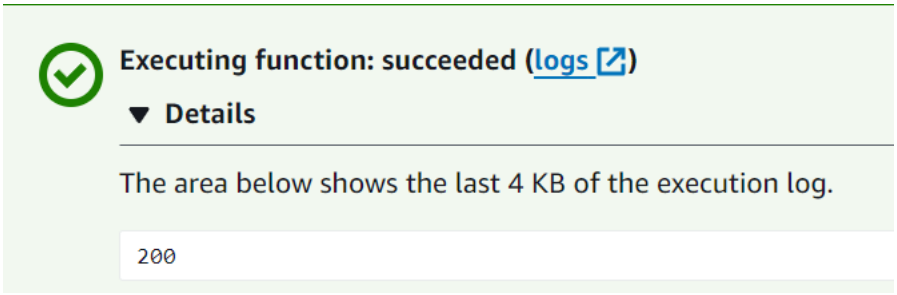
2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.
5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.

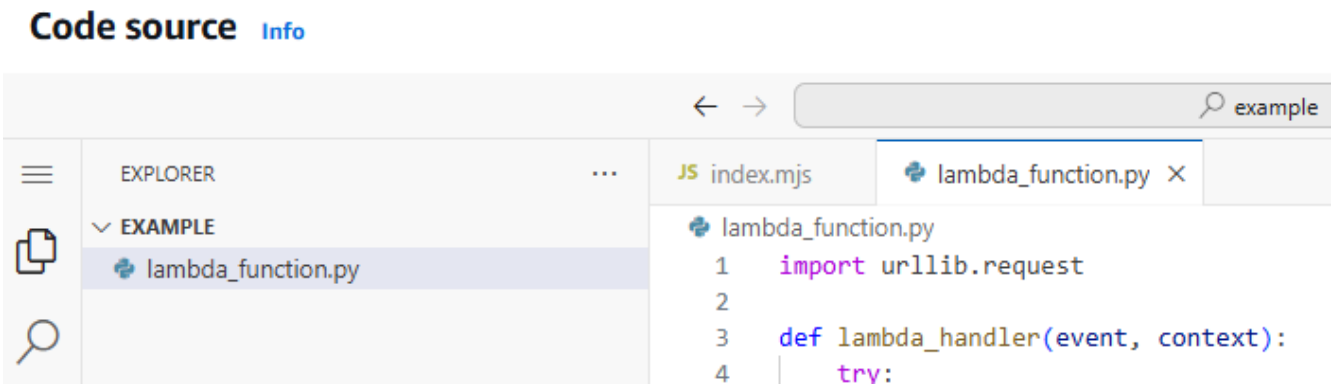


Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

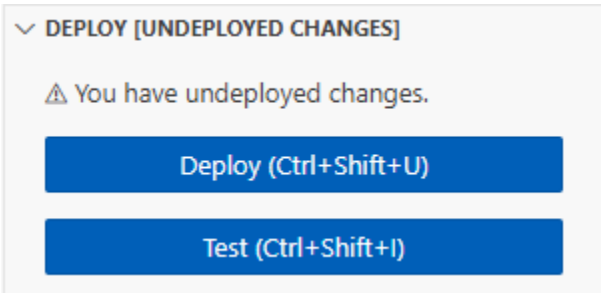
Python

1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei lambda_function.py ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.

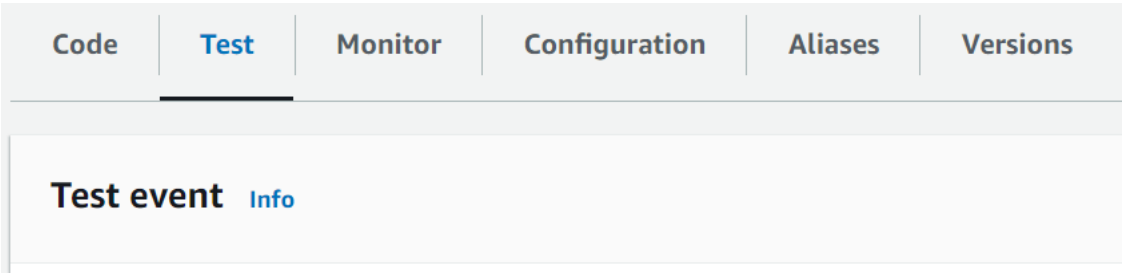



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

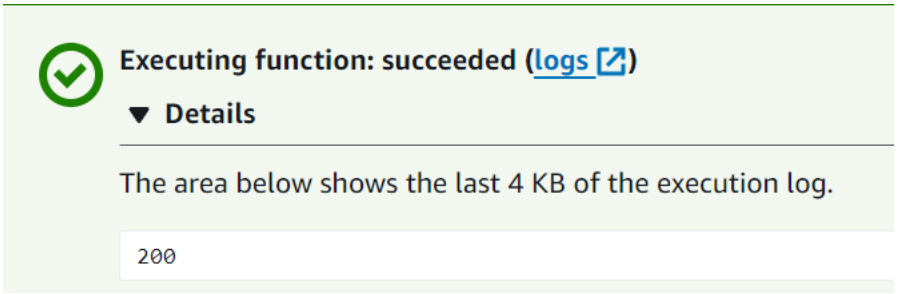
2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.
5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.



Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

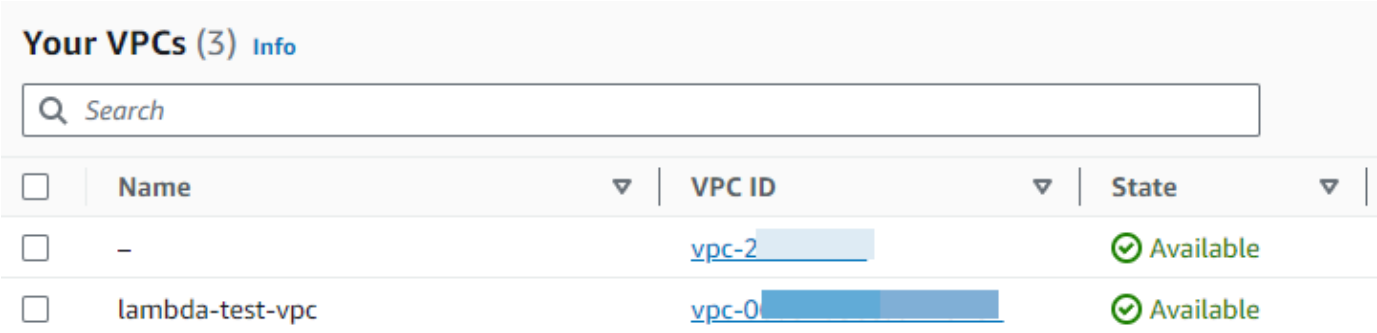
Ich habe bereits eine VPC

Wenn Sie bereits über eine VPC verfügen, aber den öffentlichen Internetzugang für eine Lambda-Funktion konfigurieren müssen, gehen Sie wie folgt vor. Bei diesem Verfahren wird davon

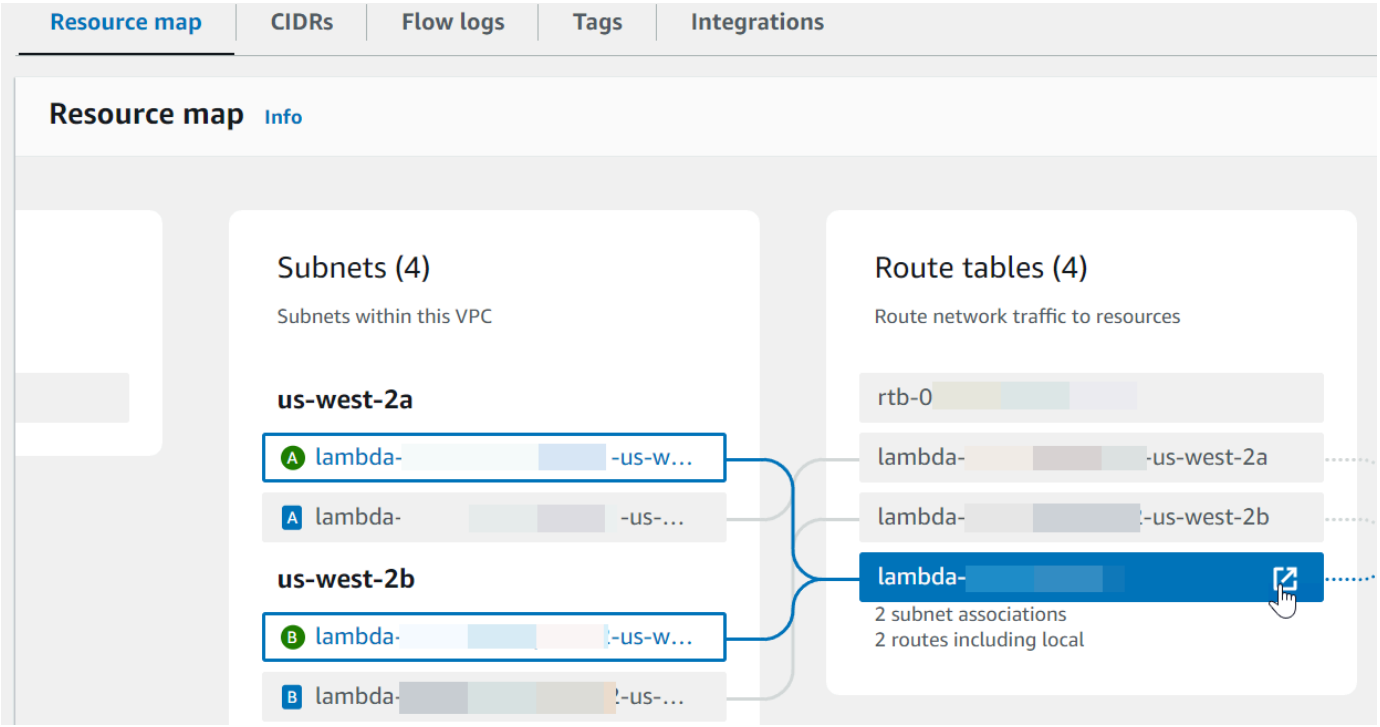
ausgegangen, dass Ihre VPC mindestens zwei Subnetze hat. Wenn Sie nicht über zwei Subnetze verfügen, finden Sie weitere Informationen unter Erstellen eines Subnetzes im Amazon-VPC-Benutzerhandbuch.

Überprüfen der Routing-Tabellen

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie die VPC ID aus.



3. Scrollen Sie nach unten zum Abschnitt Ressourcenkarte. Beachten Sie die Zuordnungen in den Routing-Tabellen. Öffnen Sie jede Routing-Tabelle, die einem Subnetz zugeordnet ist.

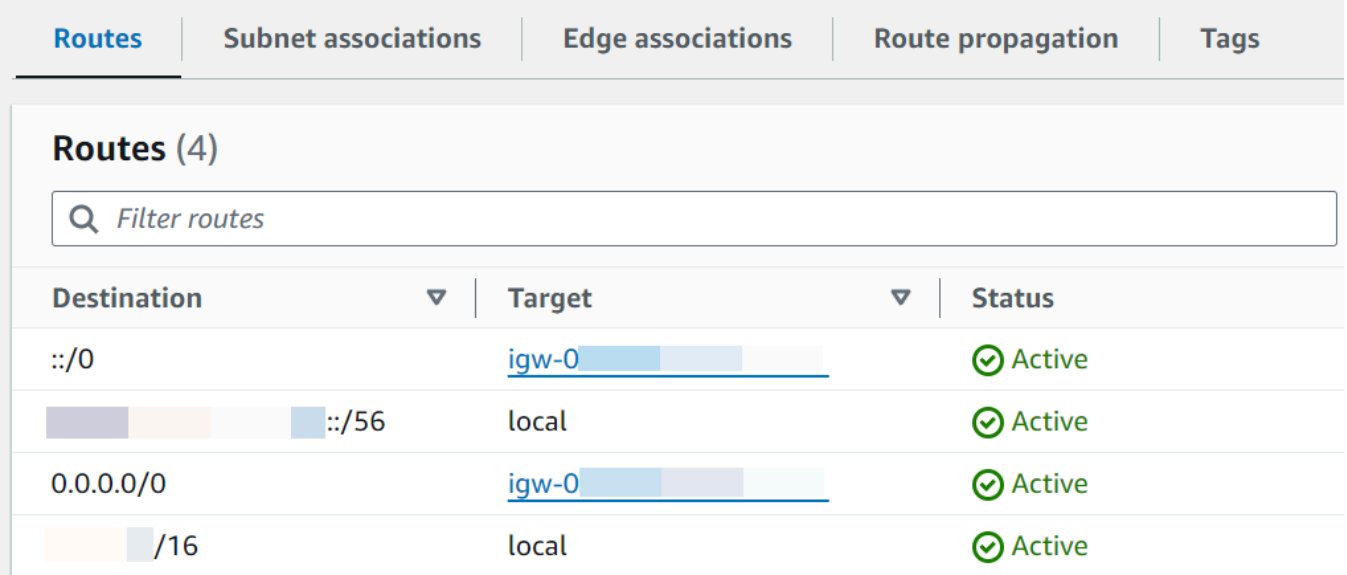


4. Scrollen Sie nach unten zur Registerkarte Routen. Überprüfen Sie die Routen, um festzustellen, ob Ihre VPC über die beiden folgenden Routing-Tabellen verfügt. Jede dieser Anforderungen muss durch eine separate Routing-Tabelle erfüllt werden.

- Internet-bound Der Datenverkehr (`0.0.0.0/0`für IPv4, `::/0` für IPv6) wird an ein Internet-Gateway () weitergeleitet. `igw-xxxxxxxxxx` Das bedeutet, dass das der Routing-Tabelle zugeordnete Subnetz ein öffentliches Subnetz ist.

> **Note**
>
> Wenn Ihr Subnetz keinen IPv6-CIDR-Block hat, wird Ihnen nur die IPv4-Route (`0.0.0.0/0`) angezeigt.

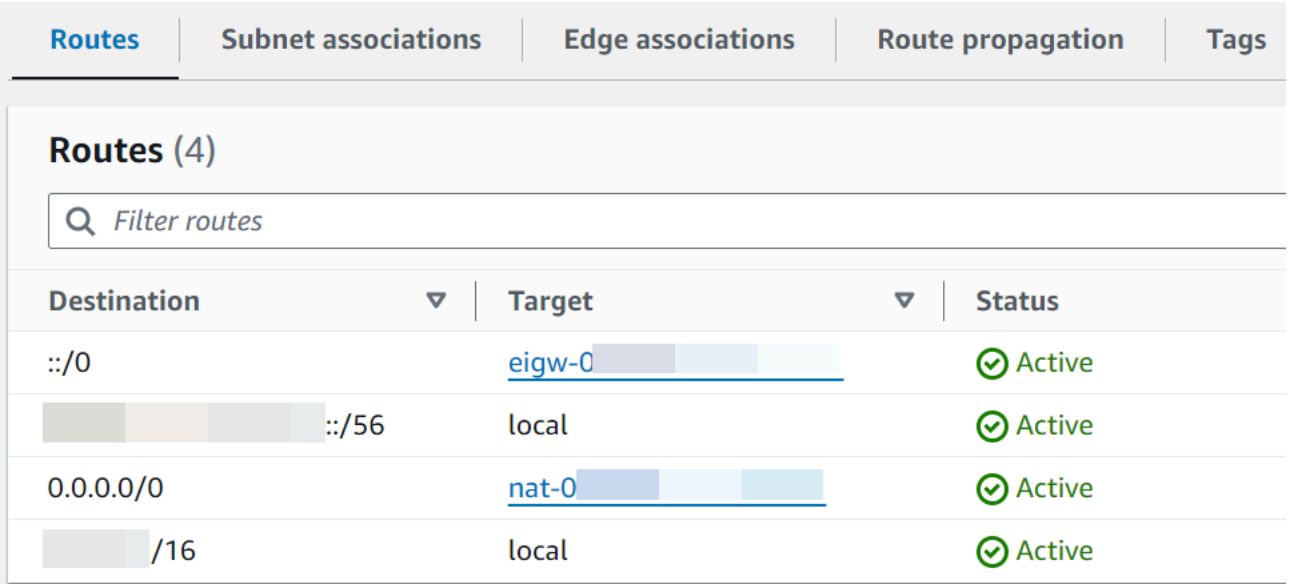
Exampleöffentliche Subnetz-Routing-Tabelle



- Internet-bound Der Datenverkehr für IPv4 (`0.0.0.0/0`) wird an ein NAT-Gateway (`nat-xxxxxxxxxx`) weitergeleitet, das einem öffentlichen Subnetz zugeordnet ist. Dies bedeutet, dass das Subnetz ein privates Subnetz ist, das über das NAT-Gateway auf das Internet zugreifen kann.

> **Note**
>
> Wenn Ihr Subnetz über einen IPv6-CIDR-Block verfügt, muss die Routing-Tabelle auch IPv6-Datenverkehr (`::/0`) an ein Internet-Gateway für ausgehenden Verkehr (`eigw-xxxxxxxxxx`) routet. Wenn Ihr Subnetz keinen IPv6-CIDR-Block hat, wird Ihnen nur die IPv4-Route (`0.0.0.0/0`) angezeigt.

Example Routing-Tabelle für private Subnetze



5. Wiederholen Sie den vorherigen Schritt, bis Sie alle Routing-Tabellen überprüft haben, die einem Subnetz in Ihrer VPC zugeordnet sind, und bestätigt haben, dass Sie über eine Routing-Tabelle mit einem Internet-Gateway und eine Routing-Tabelle mit einem NAT-Gateway verfügen.

   Wenn Sie nicht über zwei Routing-Tabellen verfügen, eine mit einer Route zu einem Internet-Gateway und eine mit einer Route zu einem NAT-Gateway, gehen Sie wie folgt vor, um die fehlenden Ressourcen und Routing-Tabelleneinträge zu erstellen.

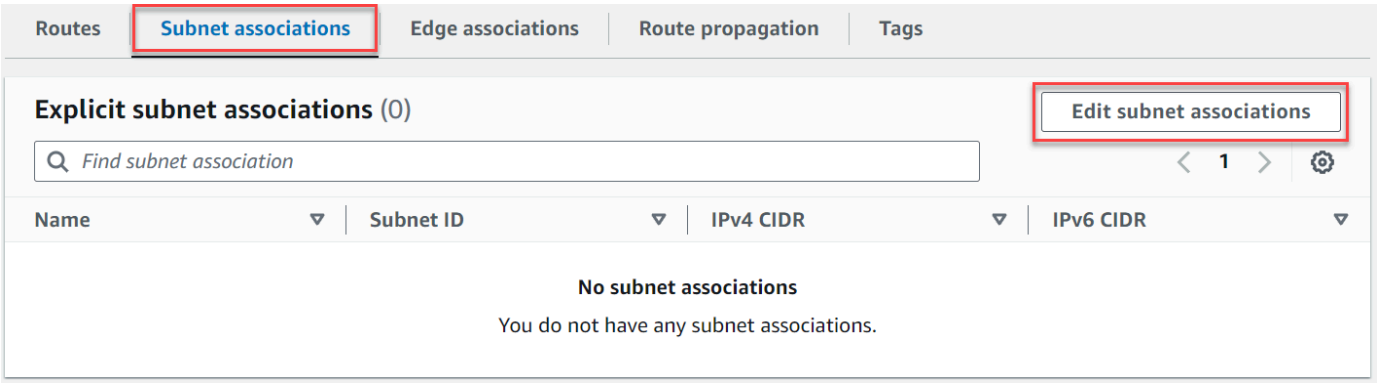
Erstellen einer Routing-Tabelle

Gehen Sie wie folgt vor, um eine Routing-Tabelle zu erstellen und diese mit einem Subnetz zu verknüpfen.

Erstellen einer benutzerdefinierten Routing-Tabelle mithilfe der Amazon-VPC-Konsole

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Route Tables (Routing-Tabellen) aus.
3. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
4. (Optional) Geben Sie bei Name einen Namen für Ihre Routing-Tabelle ein.
5. Wählen Sie unter VPC Ihre VPC aus.

6. (Optional) Sie fügen ein Tag hinzu, indem Sie Add new tag (Neuen Tag hinzufügen) auswählen und den Tag-Schlüssel und -Wert eingeben.
7. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
8. Wählen Sie auf der Registerkarte Subnet associations (Subnetzzuordnungen) die Option Edit subnet associations (Subnetzzuordnungen bearbeiten) aus.



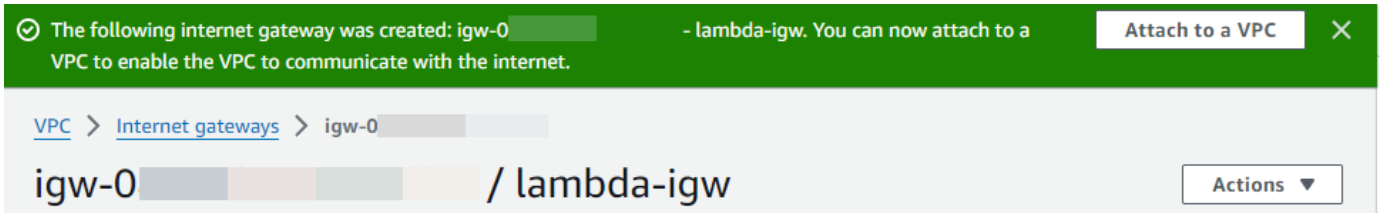
9. Aktivieren Sie das Kontrollkästchen für das Subnetz, um es der Routing-Tabelle zuzuordnen.
10. Klicken Sie auf Save associations (Zuordnungen speichern).

Ein Internet-Gateway erstellen

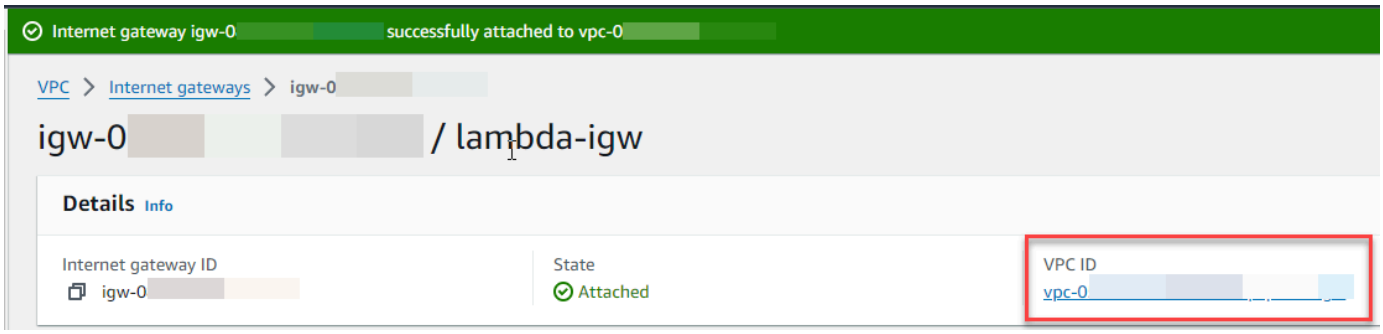
Gehen Sie wie folgt vor, um ein Internet-Gateway zu erstellen, es an Ihre VPC anzuhängen und es der Routing-Tabelle Ihres öffentlichen Subnetzes hinzuzufügen.

So erstellen Sie ein Internet-Gateway

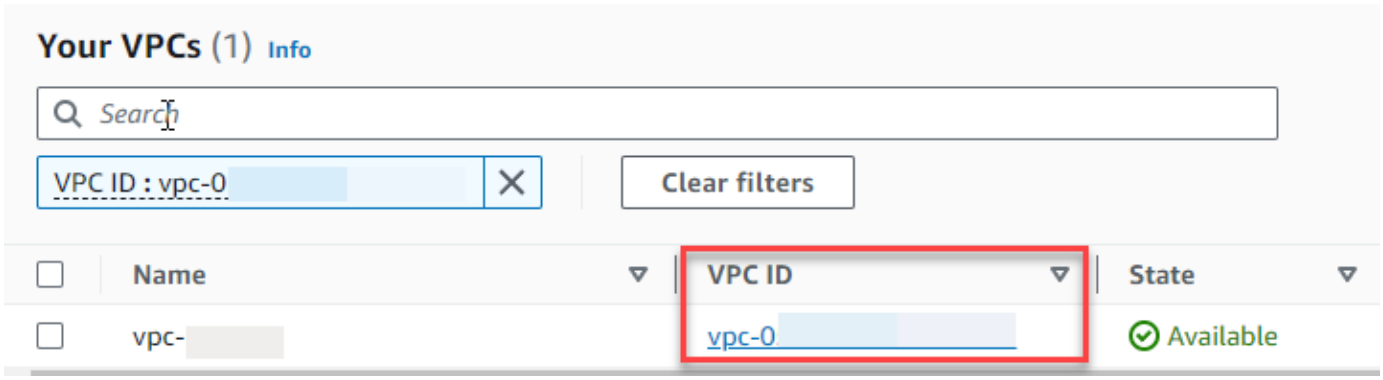
1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Internet Gateways (Internet-Gateways) aus.
3. Wählen Sie Create internet gateway (Internet-Gateway erstellen) aus.
4. (Optional) Geben Sie einen Namen für Ihr Internet-Gateway ein.
5. (Optional) Um ein Tag hinzuzufügen, wählen Sie Add new tag (Neuen Tag hinzufügen) aus und geben Sie den Schlüssel und den Wert für den Tag ein.
6. Wählen Sie Create internet gateway (Internet-Gateway erstellen) aus.
7. Wählen Sie im Banner oben auf dem Bildschirm die Option An eine VPC anhängen, wählen Sie eine verfügbare VPC aus und wählen Sie dann Internet-Gateway anhängen.



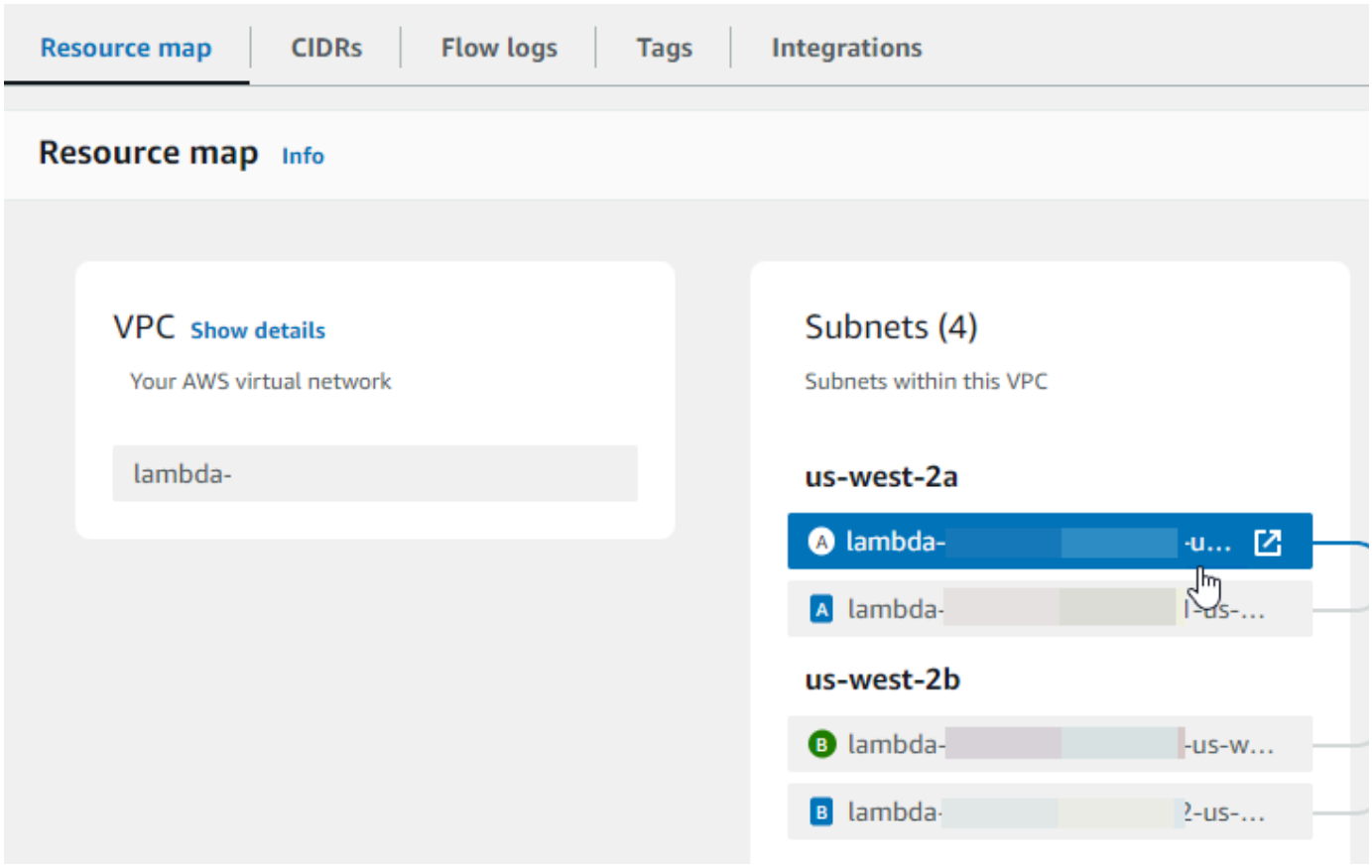
8. Wählen Sie die VPC ID aus.



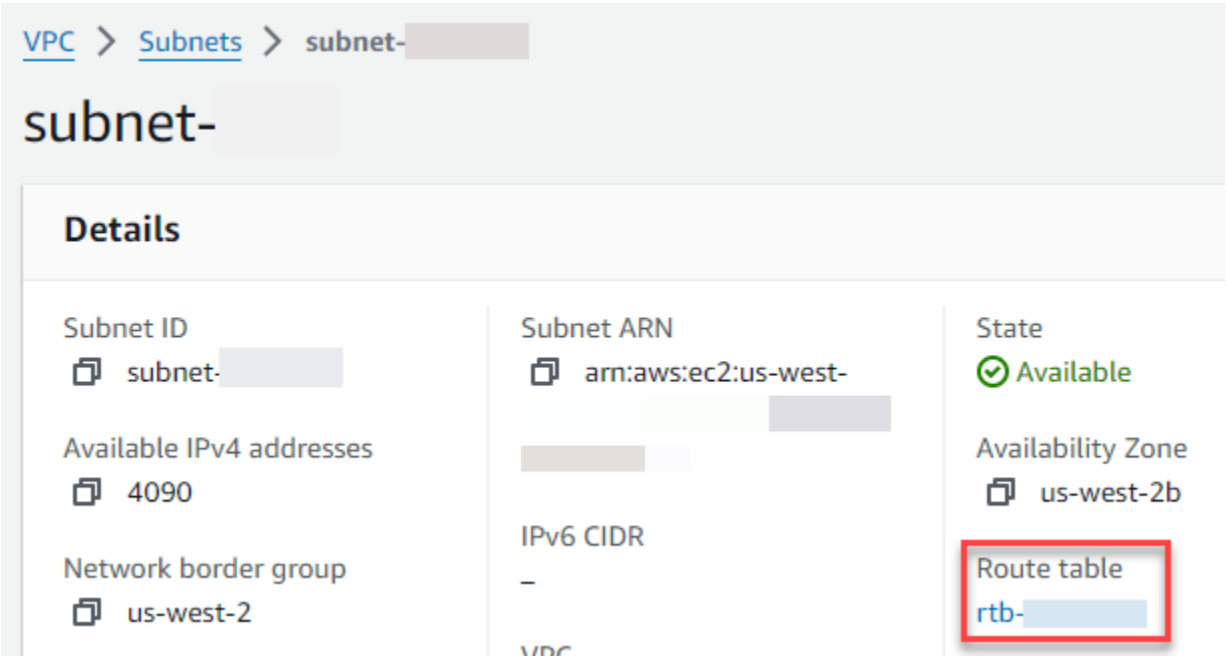
9. Wählen Sie die VPC ID, um die VPC-Detailseite zu öffnen.



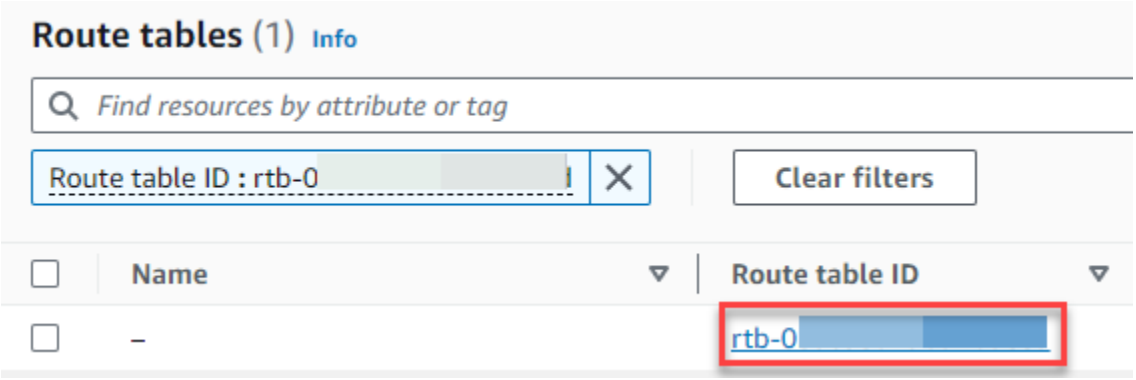
10. Blättern Sie nach unten zum Abschnitt Ressourcenzuordnung und wählen Sie dann ein Subnetz aus. Die Subnetzdetails werden auf der neuen Registerkarte angezeigt.



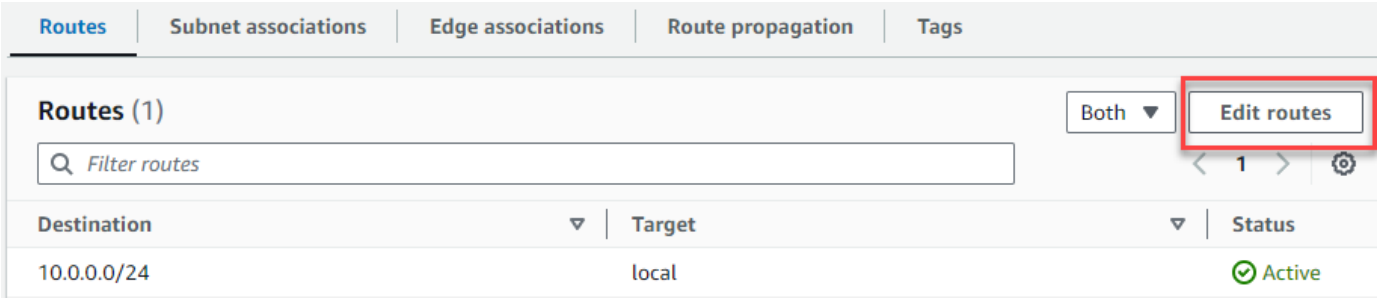
11. Wählen Sie den Link unter Routing-Tabelle aus.



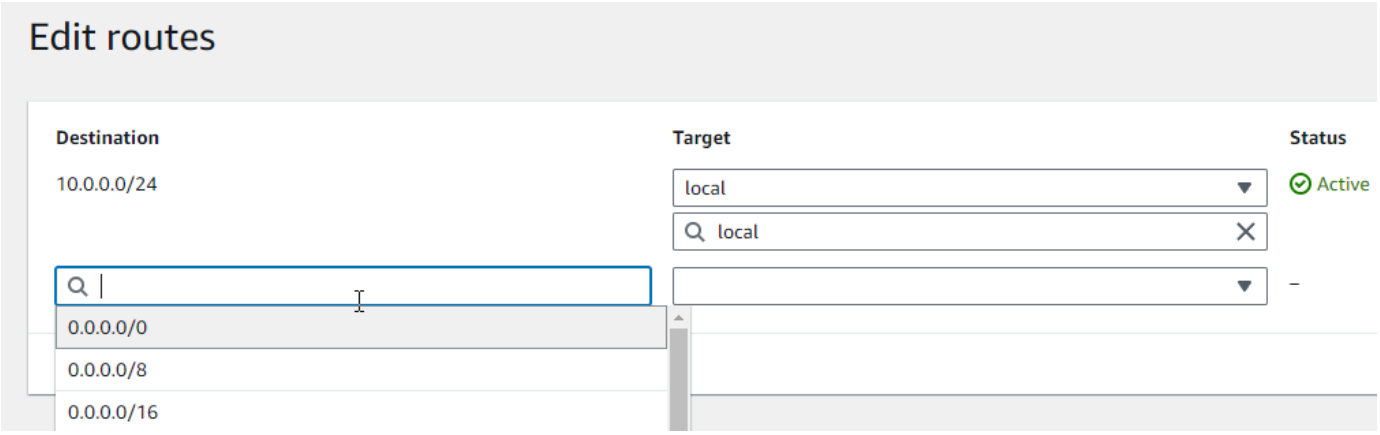
12. Wählen Sie die Routing-Tabellen-ID, um die Detailseite der Routing-Tabelle zu öffnen.



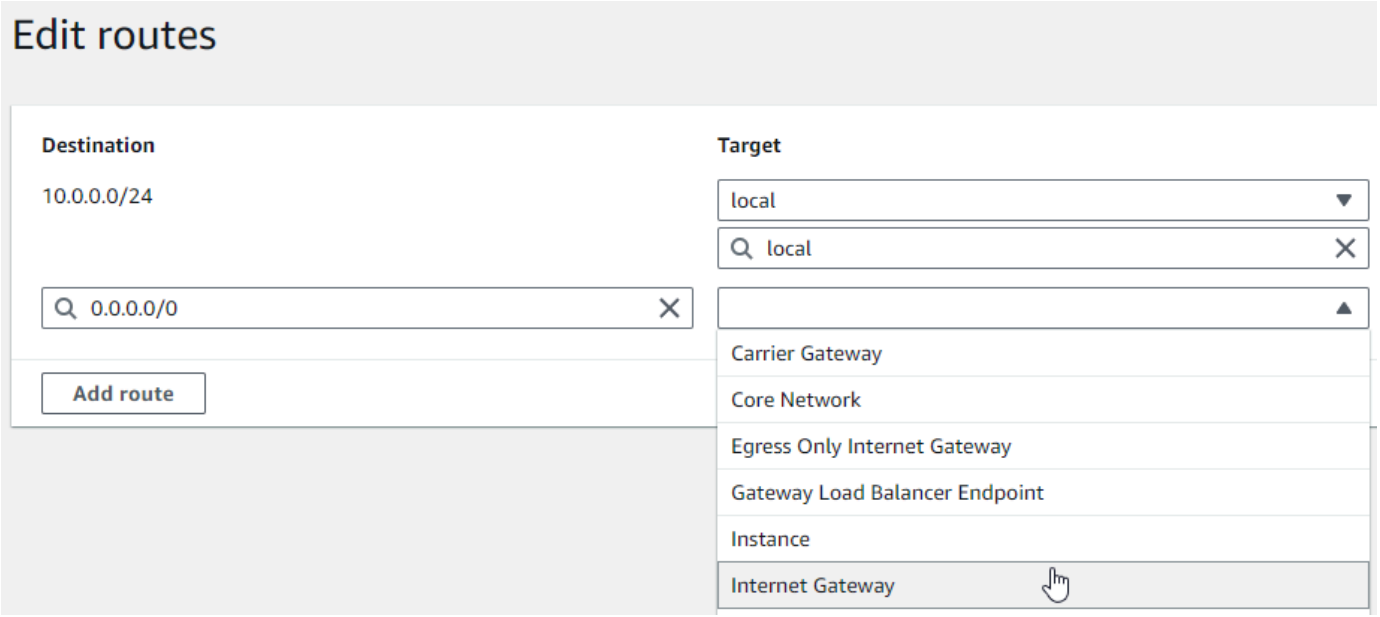
13. Wählen Sie unter Routes (Routen) Routen bearbeiten aus.



14. Wählen Sie Route hinzufügen und geben Sie sie dann `0.0.0.0/0` in das Feld Ziel ein.



15. Wählen Sie für Target Internet-Gateway und dann das Internet-Gateway aus, das Sie zuvor erstellt haben. Wenn Ihr Subnetz einen IPv6-CIDR-Block hat, müssen Sie auch eine Route für `::/0` zum gleichen Internet-Gateway hinzufügen.



16. Wählen Sie Änderungen speichern aus.

Erstellen eines NAT-Gateways

Gehen Sie wie folgt vor, um ein NAT-Gateway zu erstellen, es einem öffentlichen Subnetz zuzuordnen und es dann der Routing-Tabelle Ihres privaten Subnetzes hinzuzufügen.

Um ein NAT-Gateway zu erstellen und es einem öffentlichen Subnetz zuzuordnen

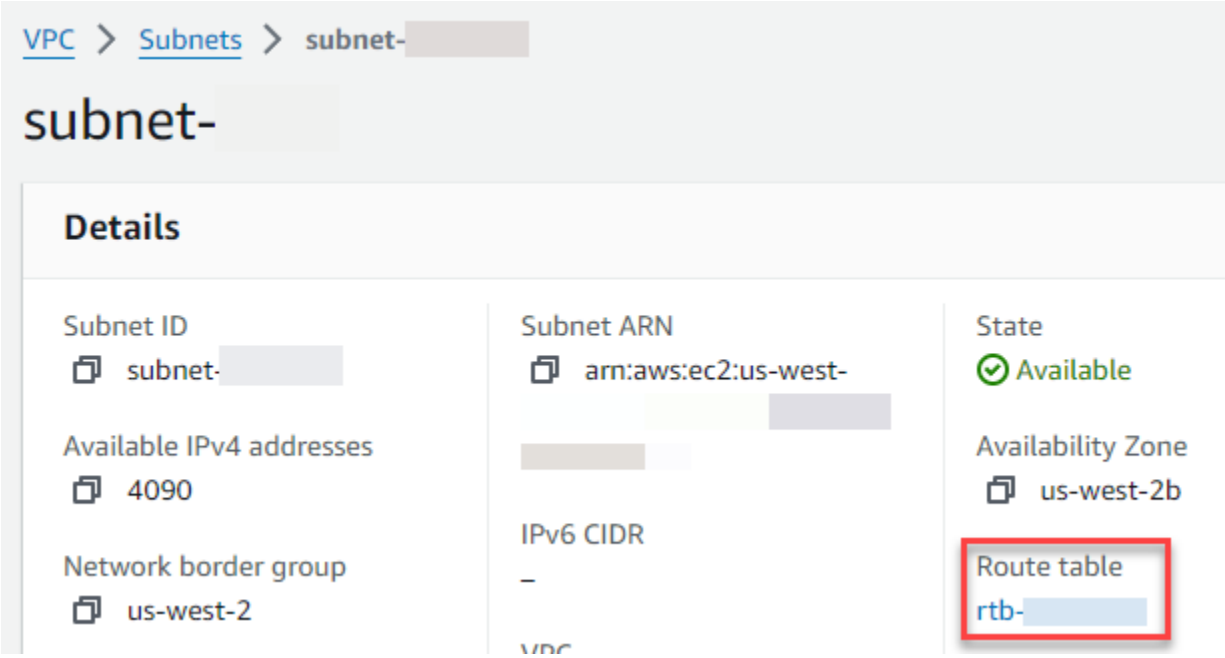
1. Klicken Sie im Navigationsbereich auf NAT-Gateways.
2. Wählen Sie NAT-Gateway erstellen aus.
3. (Optional) Geben Sie einen Namen für Ihr NAT-Gateway ein.
4. Wählen Sie unter Subnetz ein öffentliches Subnetz in Ihrer VPC. (Ein öffentliches Subnetz ist ein Subnetz, das eine direkte Route zu einem Internet-Gateway in seiner Routing-Tabelle hat).

> **Note**
>
> NAT-Gateways sind einem öffentlichen Subnetz zugeordnet, der Eintrag in der Routing-Tabelle befindet sich jedoch im privaten Subnetz.

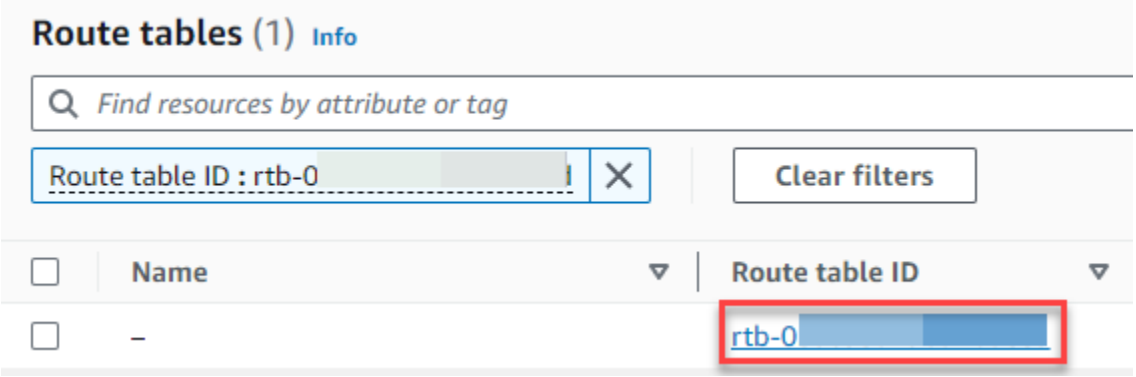
5. Wählen Sie als Elastische IP-Zuweisungs-ID eine elastische IP-Adresse oder Elastische IP zuweisen aus.
6. Wählen Sie NAT-Gateway erstellen aus.

So fügen Sie eine Route zum NAT-Gateway in der Routentabelle des privaten Subnetzes hinzu

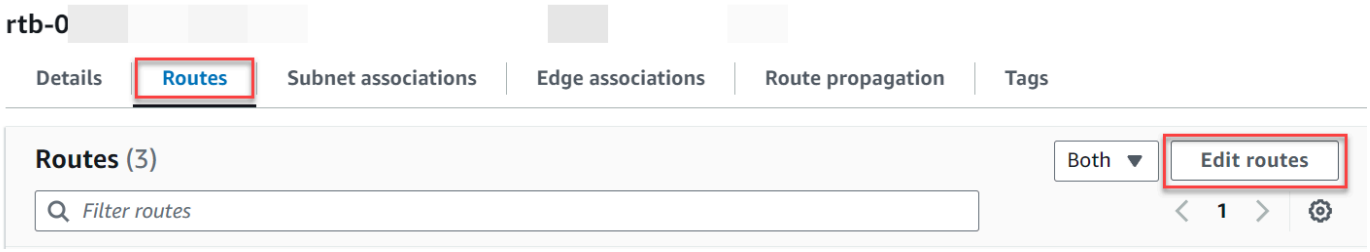
1. Wählen Sie im Navigationsbereich Subnetze aus.
2. Wählen Sie ein privates Subnetz in Ihrer VPC. (Ein privates Subnetz ist ein Subnetz, das keine Route zu einem Internet-Gateway in seiner Routing-Tabelle hat).
3. Wählen Sie den Link unter Routing-Tabelle aus.



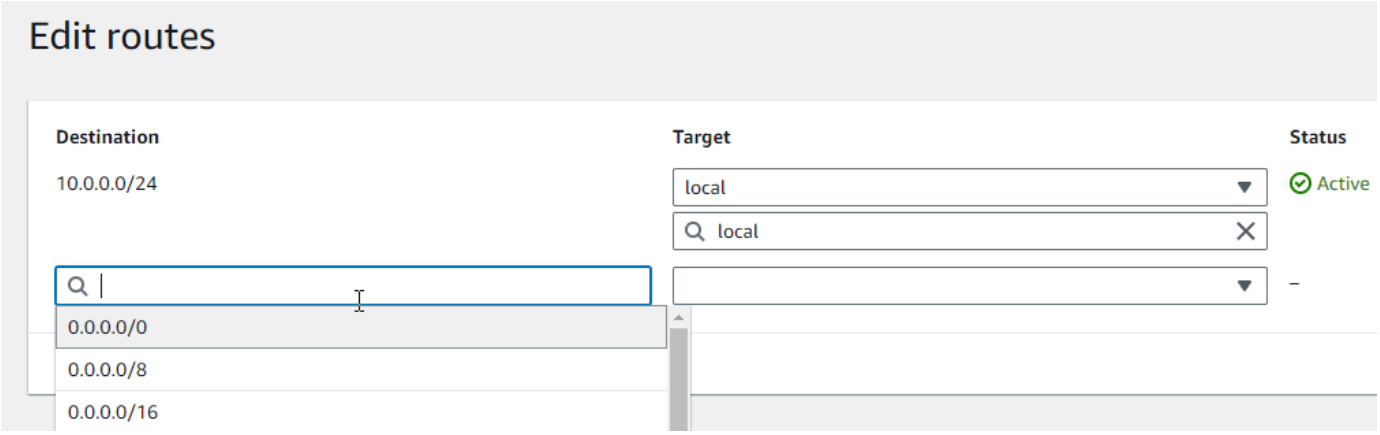
4. Wählen Sie die Routing-Tabellen-ID, um die Detailseite der Routing-Tabelle zu öffnen.



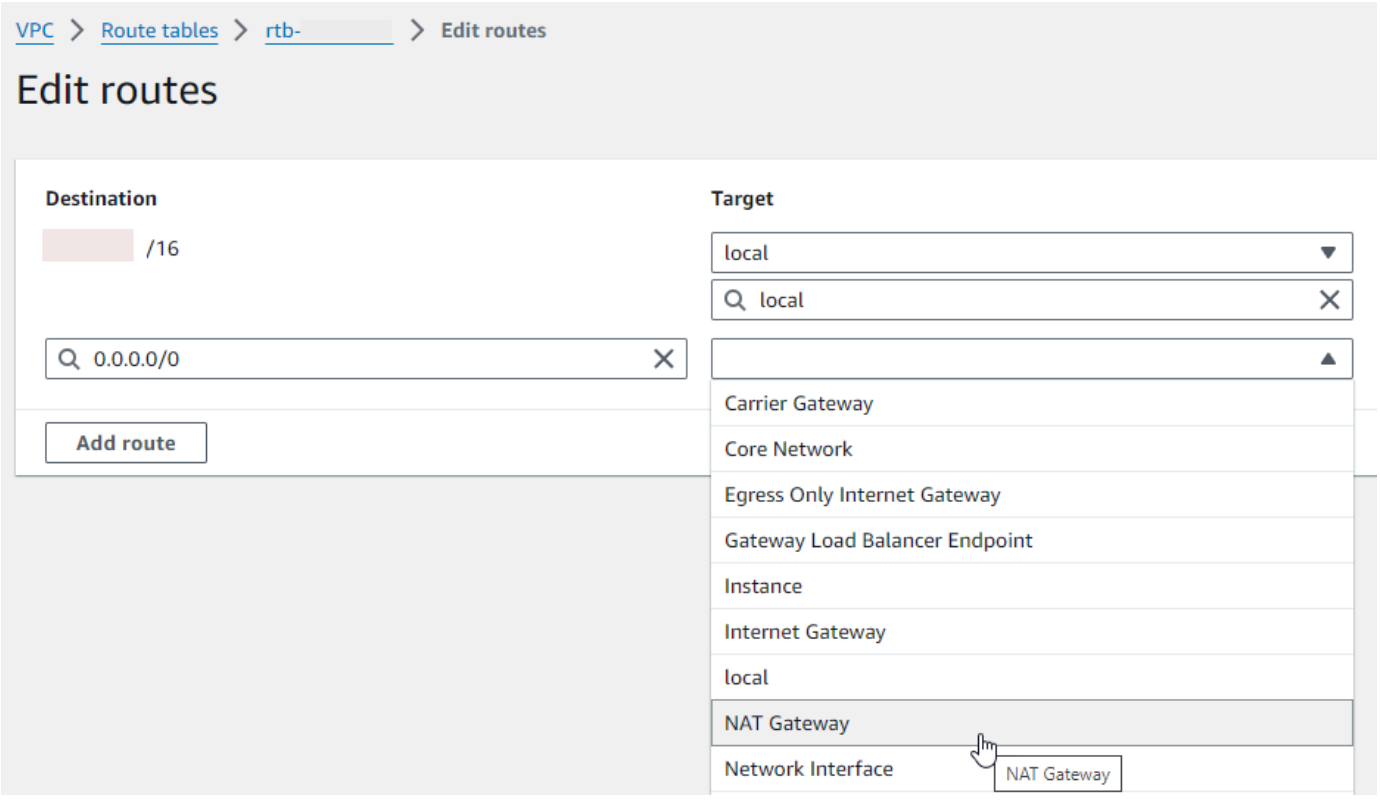
5. Scrollen Sie nach unten und wählen Sie die Registerkarte Routes (Routen) auf Edit routes (Routen bearbeiten).



6. Wählen Sie Route hinzufügen und geben Sie sie dann `0.0.0.0/0` in das Feld Ziel ein.



7. Wählen Sie für Target NAT-Gateway und dann das NAT-Gateway aus, das Sie zuvor erstellt haben.



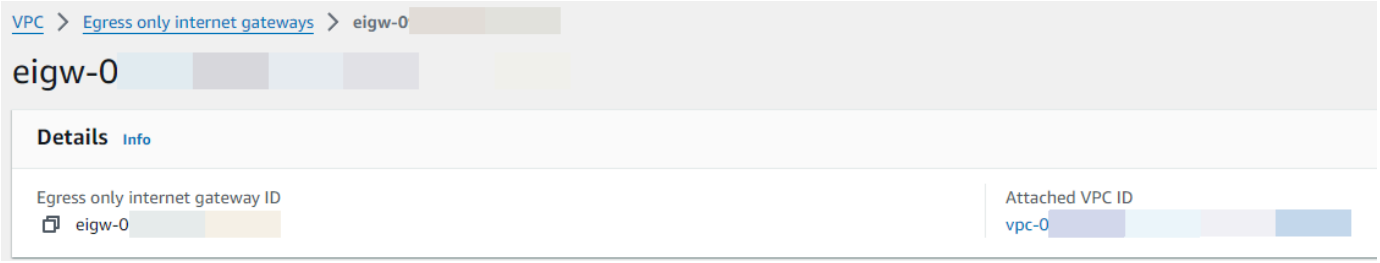
8. Wählen Sie Änderungen speichern aus.

Erstellen eines Internet-Gateways nur für ausgehenden Verkehr (nur IPv6)

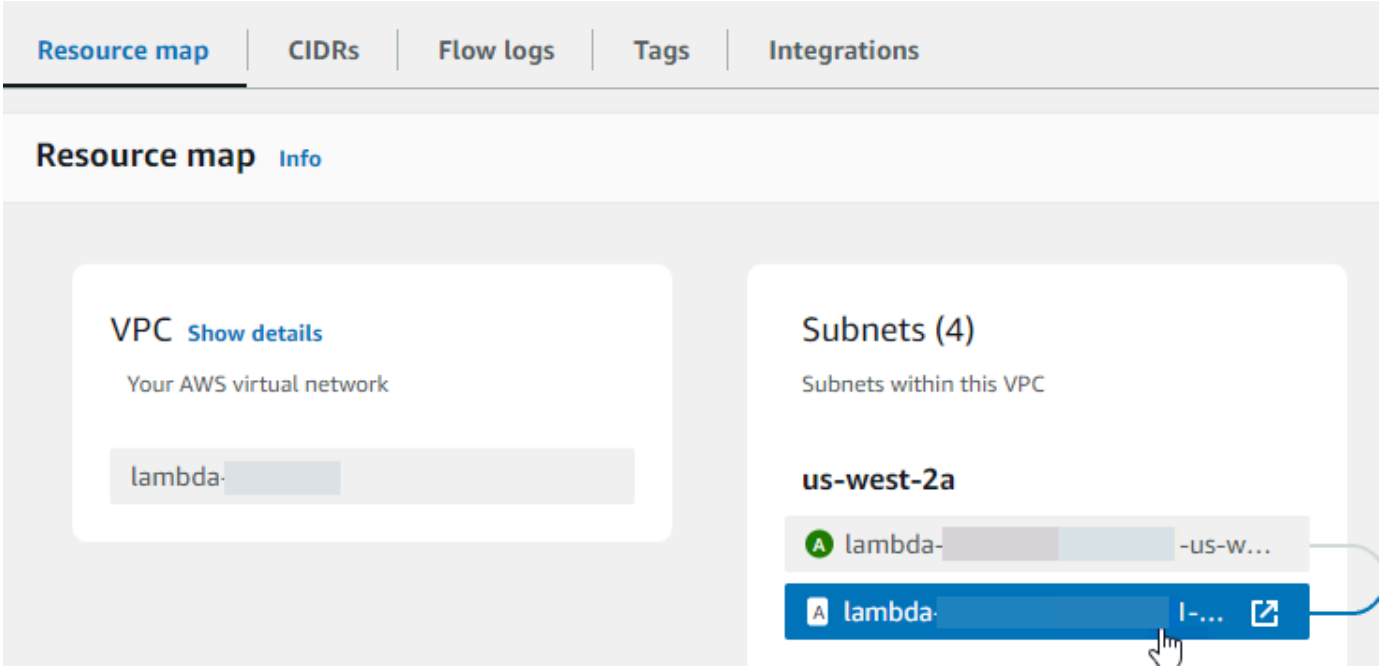
Führen Sie die folgenden Schritte aus, um ein reines Egress-Internet-Gateway zu erstellen und es zur Routing-Tabelle Ihres privaten Subnetzes hinzuzufügen.

So erstellen Sie ein Internet-Gateway nur für ausgehenden Verkehr

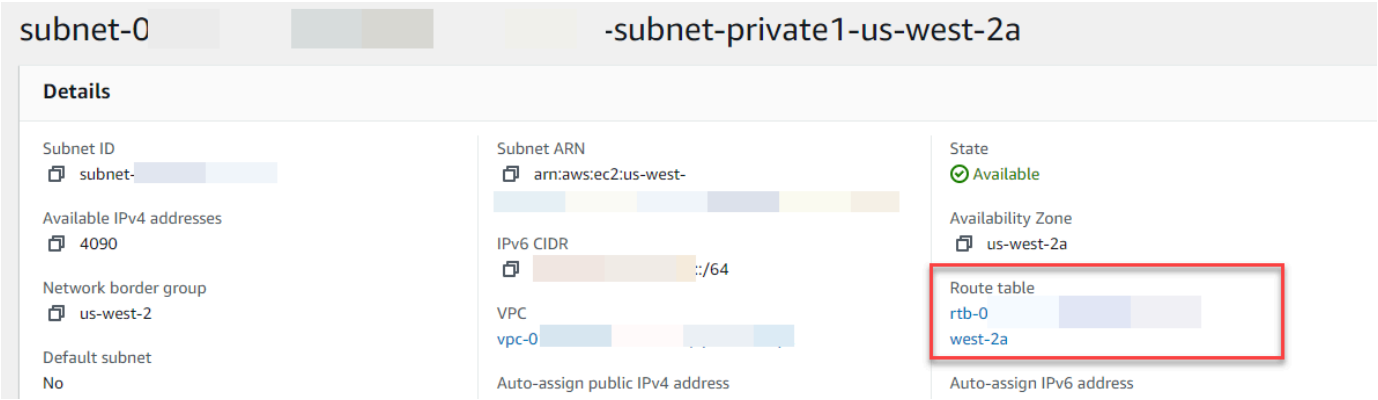
1. Wählen Sie im Navigationsbereich Egress-only Internet-Gateways aus.
2. Klicken Sie auf Internet-Gateway für ausgehenden Datenverkehr erstellen.
3. (Optional) Geben Sie einen Name ein.
4. Wählen Sie die VPC aus, in der Sie das Internet-Gateway für ausgehenden Verkehr erstellen möchten.
5. Klicken Sie auf Internet-Gateway für ausgehenden Datenverkehr erstellen.
6. Wählen Sie den Link unter Angehängte VPC-ID aus.



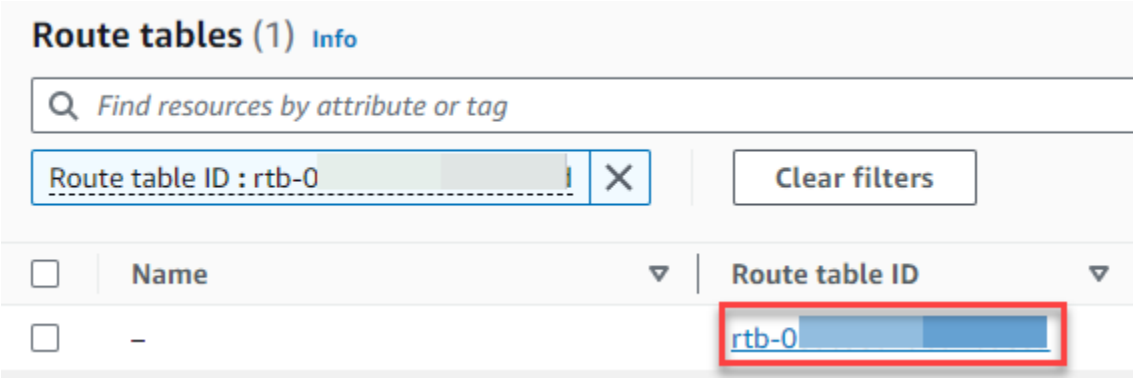
7. Wählen Sie den Link unter VPC ID, um die VPC-Detailseite zu öffnen.
8. Blättern Sie nach unten zum Abschnitt Ressourcenzuordnung und wählen Sie dann ein privates Subnetz aus. (Ein privates Subnetz ist ein Subnetz, das keine Route zu einem Internet-Gateway in seiner Routing-Tabelle hat). Die Subnetzdetails werden auf der neuen Registerkarte angezeigt.



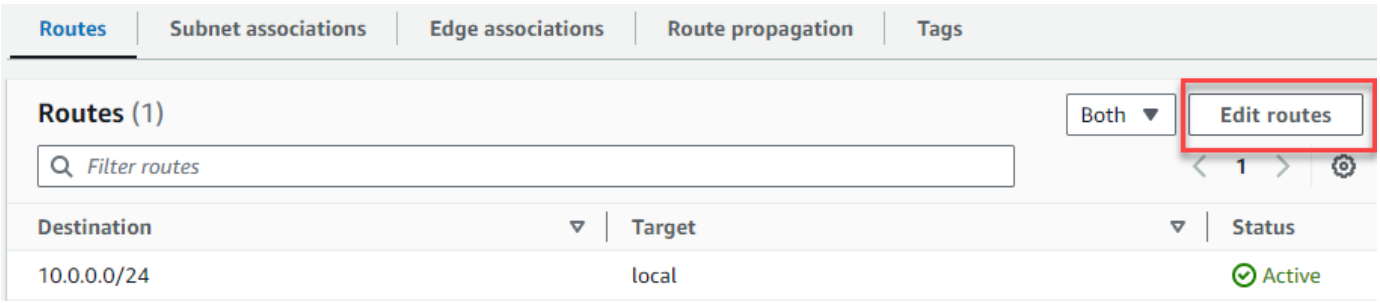
9. Wählen Sie den Link unter Routing-Tabelle aus.



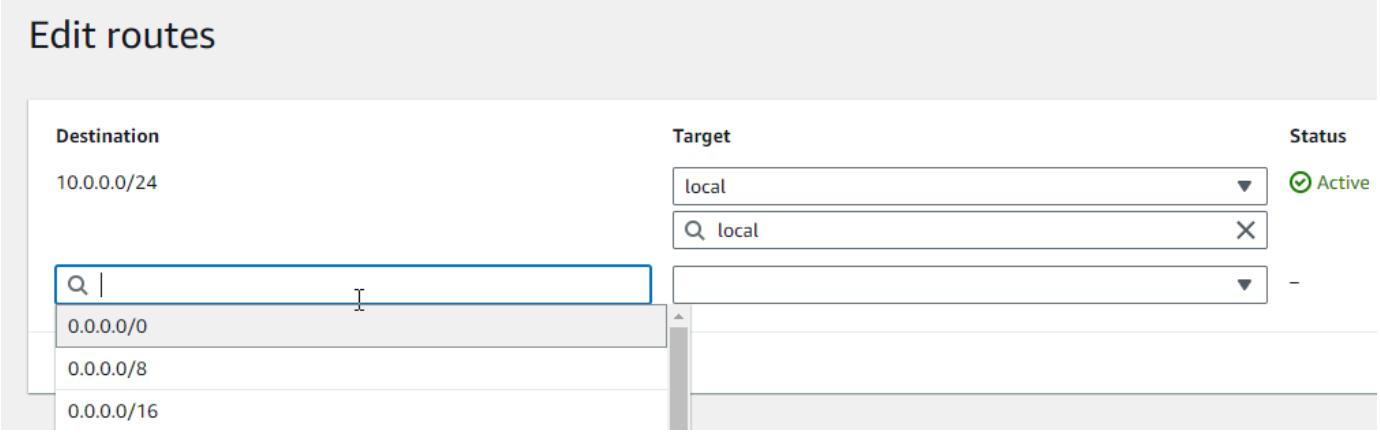
10. Wählen Sie die Routing-Tabellen-ID, um die Detailseite der Routing-Tabelle zu öffnen.



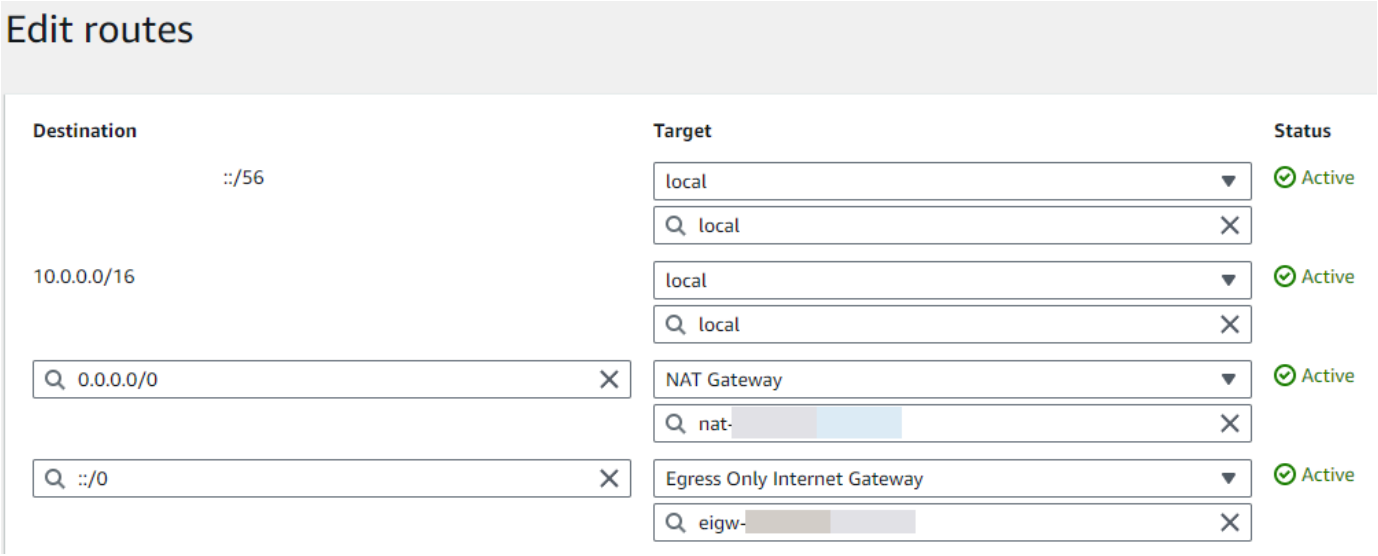
11. Wählen Sie unter Routes (Routen) Routen bearbeiten aus.



12. Wählen Sie Route hinzufügen und geben Sie sie dann `::/0` in das Feld Ziel ein.



13. Wählen Sie für Target Nur Internet-Gateway für den Ausgang und dann das Internet-Gateway aus, das Sie zuvor erstellt haben.



14. Wählen Sie Änderungen speichern aus.

Konfigurieren der Lambda-Funktion

So konfigurieren Sie eine VPC bei der Funktionserstellung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Geben Sie unter Basic information (Grundlegende Informationen) bei Function name (Funktionsname) einen Namen für Ihre Funktion ein.
4. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
5. Wählen Sie VPC aktivieren und wählen Sie dann eine VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

   > **Note**
   >
   > Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie „Event-Funktion erstellen“.

Lambda erstellt automatisch eine Ausführungsrolle mit der AWSLambdaVPCAccessExecutionRole AWS verwalteten Richtlinie. Die Berechtigungen in dieser Richtlinie sind nur zum Erstellen von Elastic Network-Schnittstellen für die VPC-Konfiguration und nicht zum Aufrufen Ihrer Funktion erforderlich. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die AWSLambdaVPCAccessExecutionRoleRichtlinie aus Ihrer Ausführungsrolle entfernen, nachdem Sie die Funktion und die VPC-Konfiguration erstellt haben. Weitere Informationen finden Sie unter Erforderliche IAM-Berechtigungen.

So konfigurieren Sie eine VPC für eine vorhandene Funktion

Um einer vorhandenen Funktion eine VPC-Konfiguration hinzuzufügen, muss die Ausführungsrolle der Funktion über die Berechtigung verfügen, elastische Netzwerkschnittstellen zu erstellen und zu verwalten. Die AWSLambdaVPCAccessExecutionRole AWS verwaltete Richtlinie umfasst die erforderlichen Berechtigungen. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die AWSLambdaVPCAccessExecutionRoleRichtlinie aus Ihrer Ausführungsrolle entfernen, nachdem Sie die VPC-Konfiguration erstellt haben.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
4. Wählen Sie unter VPC die Option Edit (Bearbeiten) aus.
5. Wählen Sie die VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

   > **Note**
   >
   > Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

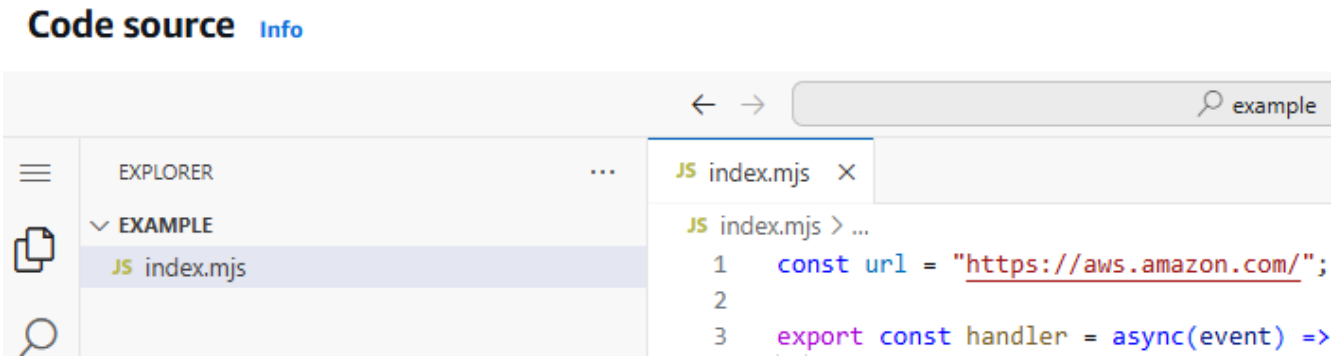
8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie Speichern.

Testen der Funktion

Verwenden Sie den folgenden Beispielcode, um zu bestätigen, dass Ihre VPC-connected Funktion das öffentliche Internet erreichen kann. Bei Erfolg gibt der Code einen `200`-Statuscode zurück. Ist dies nicht der Fall, wird die Funktion abgebrochen.

Node.js

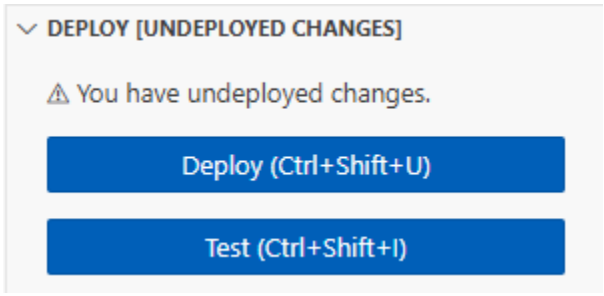
1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei index.mjs ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.



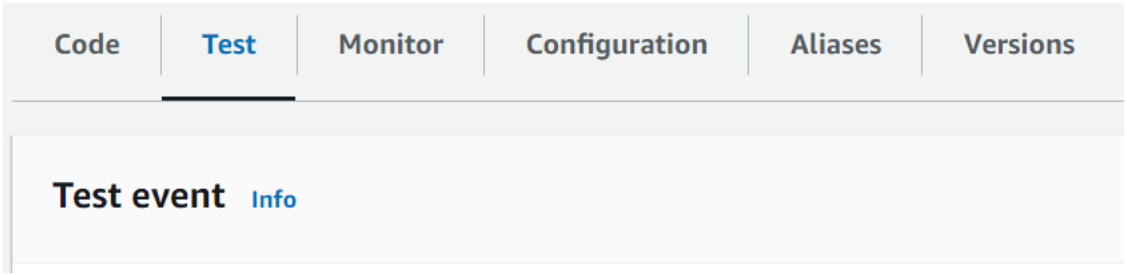
Example— HTTP-Anfrage mit async/await

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

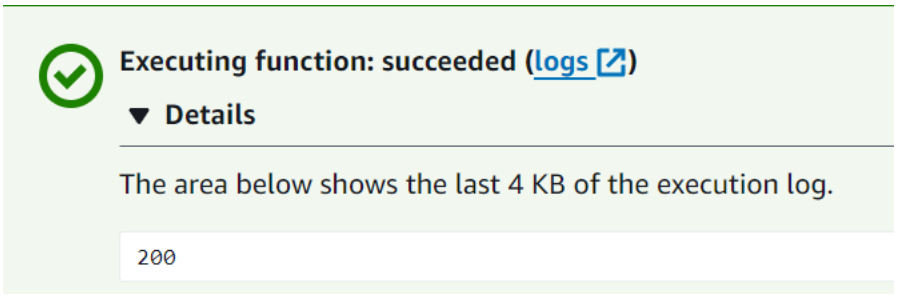
2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.
5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.

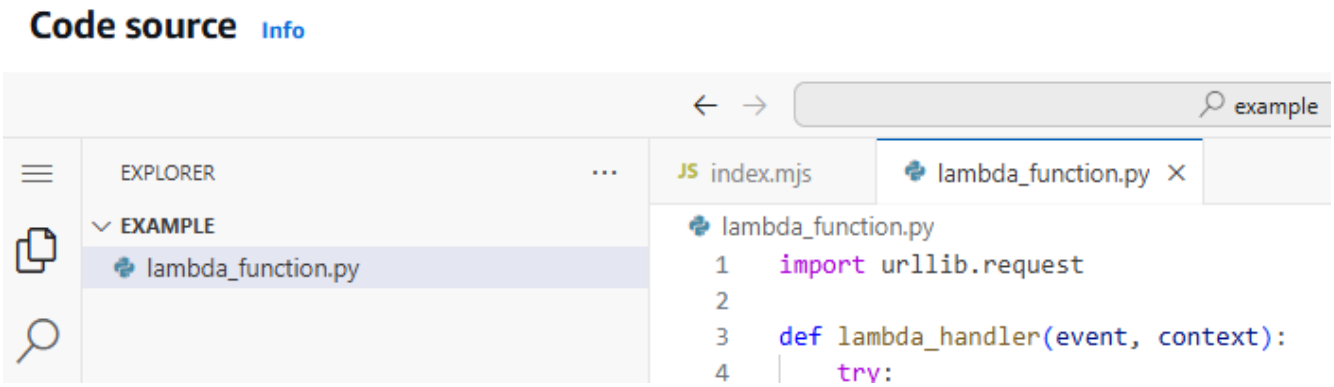


Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

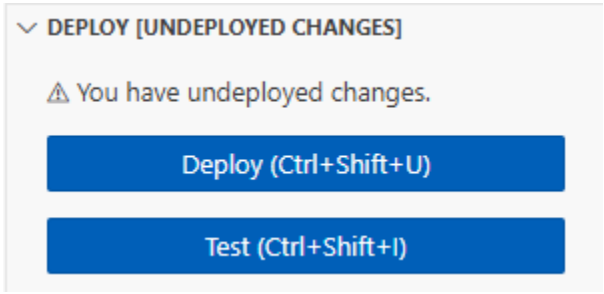
Python

1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei lambda_function.py ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.

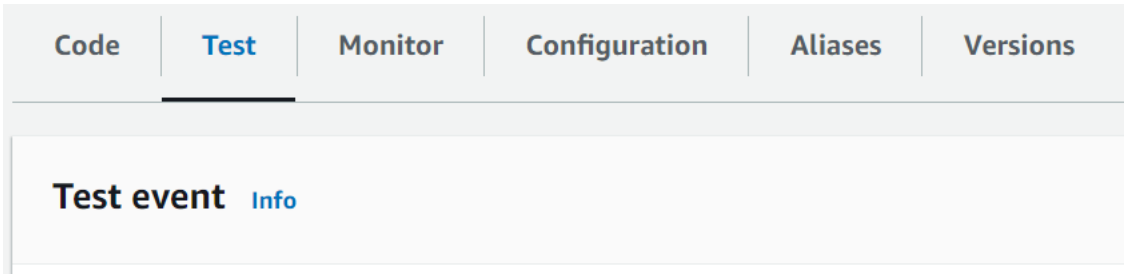



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:

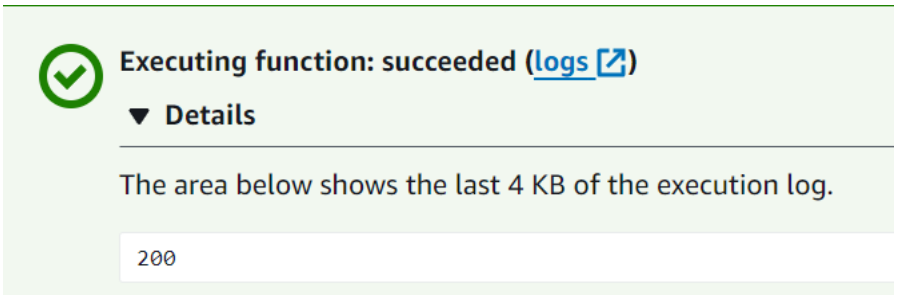


3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.

5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.



Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Verbinden von VPC-Endpunkten von eingehenden Schnittstellen für Lambda

Wenn Sie Amazon Virtual Private Cloud (Amazon VPC) zum Hosten Ihrer AWS Ressourcen verwenden, können Sie eine Verbindung zwischen Ihrer VPC und Lambda herstellen. Sie können mit dieser Verbindung Ihre Lambda-Funktion aufrufen, ohne das öffentliche Internet überqueren.

Um eine private Verbindung zwischen Ihrer VPC und Lambda herzustellen, erstellen Sie einen Schnittstellen-VPC-Endpunkt. Schnittstellenendpunkte werden von unterstütz AWS PrivateLink. Mit können Sie privat ohne Internet-Gateway AWS PrivateLink, NAT-Gerät, VPN-Verbindung oder Verbindung auf Lambda-APIs zugreifen. AWS Direct Connect Die Instances in Ihrer VPC benötigen für die Kommunikation mit Lambda-APIs keine öffentlichen IP-Adressen. Datenverkehr zwischen Ihrer VPC und Lambda verlässt das AWS -Netzwerk nicht.

Jeder Schnittstellenendpunkt wird durch eine oder mehrere Elastic Network-Schnittstellen in Ihren Subnetzen dargestellt. Eine Netzwerkschnittstelle stellt eine private IP-Adresse bereit, die als Einstiegspunkt für den Datenverkehr zu Lambda dient.

Abschnitte

- Überlegungen für Lambda-Schnittstellenendpunkte
- Erstellen eines Schnittstellenendpunkts für Lambda
- Erstellen einer Richtlinie des Schnittstellenendpunkts für Lambda

Überlegungen für Lambda-Schnittstellenendpunkte

Bevor Sie einen Schnittstellenendpunkt für Lambda einrichten, lesen Sie unbedingt die Eigenschaften und Einschränkungen des Schnittstellenendpunkts im Amazon-VPC-Benutzerhandbuch.

Sie können alle Lambda-API-Operationen von Ihrer VPC aus aufrufen. Sie können die Lambda-Funktion beispielsweise aufrufen, indem Sie die `Invoke`-API aus Ihrer VPC aufrufen. Die vollständige Liste der Lambda-APIs finden Sie unter Aktionen in der Lambda-API-Referenz.

`use1-az3` ist eine Region mit begrenzter Kapazität für Lambda-VPC-Funktionen. Sie sollten in dieser Availability Zone keine Subnetze für Ihre Lambda-Funktionen verwenden, da dies im Falle eines Ausfalls zu einer verringerten zonalen Redundanz führen kann.

Keep-alive für persistente Verbindungen

Lambda löscht Leerlaufverbindungen im Laufe der Zeit. Daher müssen Sie eine Keep-Alive-Direktive verwenden, um dauerhafte Verbindungen zu pflegen. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie im AWS SDK für JavaScript Entwicklerhandbuch unter Wiederverwenden von Verbindungen mit Keep-Alive Node.js in.

Überlegungen zur Abrechnung

Es fallen keine zusätzlichen Kosten für den Zugriff auf eine Lambda-Funktion über einen Schnittstellen-Endpunkt an. Weitere Informationen über Lambda-Preise finden Sie unter AWS Lambda -Preise.

Die Standardpreise für AWS PrivateLink gelten für Schnittstellenendpunkte für Lambda. Ihrem AWS Konto wird jede Stunde in Rechnung gestellt, in der ein Schnittstellenendpunkt in jeder Availability Zone bereitgestellt wird, sowie für Daten, die über den Schnittstellenendpunkt verarbeitet werden. Weitere Informationen zu Preisen für Schnittstellenendpunkte finden Sie unter AWS PrivateLink - Preise.

Überlegungen zu VPC Peering

Sie können andere VPCs mit Schnittstellenendpunkten über VPC-Peering mit der VPC verbinden. VPC-Peering ist eine Netzwerkverbindung zwischen zwei VPCs. Sie können eine VPC-Peering-Verbindung zwischen Ihren eigenen beiden VPCs oder mit einer VPC in einem anderen AWS -Konto herstellen. Die VPCs können sich auch in zwei verschiedenen Regionen befinden. AWS

Der Datenverkehr zwischen Peer-VPCs verbleibt im AWS Netzwerk und durchquert nicht das öffentliche Internet. Sobald die VPCs miteinander verbunden sind, können Ressourcen wie Amazon Elastic Compute Cloud (Amazon EC2) -Instances, Amazon Relational Database Service (Amazon RDS) -Instances oder VPC-enabled Lambda-Funktionen in beiden VPCs über Schnittstellenendpunkte, die in einer der VPCs erstellt wurden, auf die Lambda-API zugreifen.

Erstellen eines Schnittstellenendpunkts für Lambda

Sie können einen Schnittstellenendpunkt für Lambda entweder mit der Amazon VPC-Konsole oder mit () erstellen. AWS Command Line Interface AWS CLI Weitere Informationen finden Sie unter Erstellung eines Schnittstellenendpunkts im Benutzerhandbuch für Amazon VPC.

So erstellen Sie einen Schnittstellenendpunkt für Lambda (Konsole)

1. Öffnen Sie die Seite Endpunkte der Amazon-VPC-Konsole.
2. Klicken Sie auf Create Endpoint.
3. Stellen Sie sicher, dass für Service-Kategorie die Option AWS-Services ausgewählt ist.
4. Wählen Sie com.amazonaws als Servicename aus. ***region***.lambda. Stellen Sie sicher, dass als Typ die Option Schnittstelle gewählt ist.
5. Auswählen von VPC und Subnetzen
6. Um ein privates DNS für den Schnittstellenendpunkt zu aktivieren, aktivieren Sie das Kontrollkästchen für Name eines DNS aktivieren. Wir empfehlen die Aktivierung privater DNS-Namen für Ihren VPC-Endpunkt für AWS-Services. Dadurch wird sichergestellt, dass Anfragen, die die Endpunkte des öffentlichen Dienstes verwenden, wie z. B. Anfragen, die über ein AWS SDK gestellt werden, an Ihren VPC-Endpunkt weitergeleitet werden.
7. Wählen Sie unter Security group (Sicherheitsgruppe) eine oder mehrere Sicherheitsgruppen aus.
8. Wählen Sie Endpunkt erstellen aus.

Um die private DNS-Option verwenden zu können, müssen Sie `enableDnsHostnames` und `enableDnsSupportattributes` Ihrer VPC festlegen. Weitere Informationen finden Sie unter Anzeigen und Aktualisieren der DNS-Unterstützung für Ihre VPC im Amazon-VPC-Benutzerhandbuch. Wenn Sie einen privaten DNS für den Schnittstellenendpunkt aktivieren, können Sie mittels seines standardmäßigen DNS-Namen für die Region, beispielsweise `lambda.us-east-1.amazonaws.com`, API-Anforderungen an Lambda senden. Weitere Service-Endpunkte finden Sie unter Service-Endpunkte und Kontingente im Allgemeine AWS-Referenz.

Weitere Informationen finden Sie unter Zugriff auf einen Service über einen Schnittstellenendpunkt im Benutzerhandbuch für Amazon VPC.

Informationen zum Erstellen und Konfigurieren eines Endpunkts mit Hilfe CloudFormation von AWS: :EC2: :VPCEndpoint finden Sie in der Ressource AWS: :VPCEndpoint im Benutzerhandbuch. AWS CloudFormation

Um einen Schnittstellenendpunkt für Lambda zu erstellen (AWS CLI)

Verwenden Sie den Befehl create-vpc-endpoint und geben Sie die VPC-ID, den VPC-Endpunkttyp (Schnittstelle), den Dienstnamen, Subnetze, die den Endpunkt verwenden, und Sicherheitsgruppen an, die den Netzwerkschnittstellen des Endpunkts zugeordnet werden sollen. Beispiel:

```
aws ec2 create-vpc-endpoint
  --vpc-id vpc-ec43eb89
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.us-east-1.lambda
  --subnet-id subnet-abababab
  --security-group-id sg-1a2b3c4d
```

Erstellen einer Richtlinie des Schnittstellenendpunkts für Lambda

Wenn Sie steuern möchten, wer Ihren Schnittstellenendpunkt verwenden kann und auf welche Lambda-Funktionen der Benutzer zugreifen kann, können Sie eine Endpunktrichtlinie an Ihren Endpunkt anfügen. Die Richtlinie gibt die folgenden Informationen an:

- Prinzipal, der die Aktionen ausführen kann
- Die Aktionen, die der Prinzipal ausführen kann.
- Die Ressourcen, auf denen der Prinzipal Aktionen ausführen kann.

Weitere Informationen finden Sie unter Steuerung des Zugriffs auf Services mit VPC-Endpunkten im Amazon VPC Benutzerhandbuch.

Beispiel: Schnittstellenendpunktrichtlinie für Lambda-Aktionen

Im Folgenden finden Sie ein Beispiel für eine Endpunktrichtlinie für Lambda. Bei Anfügung an einen Endpunkt ermöglicht diese Richtlinie dem Benutzer `MyUser` die Funktion `my-function` aufzurufen.

> **Note**
>
> Sie müssen sowohl den qualifizierten als auch den nicht qualifizierten Funktions-ARN in die Ressource aufnehmen.

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
         "Effect":"Allow",
```

```
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
      }
   ]
}
```

Informationen dazu, wie Sie Ihrer Funktion Zugriff auf Ressourcen in einer VPC gewähren, die zu einer anderen VPC gehört, finden Sie unter. AWS-Kontothe section called “Funktionen an Ressourcen in einem anderen Konto anhängen”

Konfiguration des Dateisystemzugriffs für Lambda Event Functions

Sie können eine Lambda-Funktion konfigurieren, um ein Dateisystem in ein lokales Verzeichnis einzuhängen. Lambda unterstützt die folgenden Dateisystemtypen:

- Amazon Elastic File System (Amazon EFS) — Serverloses Dateisystem, das sich automatisch an Ihre Workloads anpasst.
- Amazon S3 Files — Serverloses Dateisystem zum Mounten Ihres Amazon S3 S3-Buckets. Amazon S3 Files ermöglicht den Zugriff auf Ihre Amazon S3 S3-Objekte als Dateien mithilfe von standardmäßigen Dateisystemoperationen wie Lesen und Schreiben auf dem lokalen Bereitstellungspfad.

> **Note**
>
> Eine Lambda-Funktion kann entweder Amazon EFS oder Amazon S3 Files verwenden, aber nicht beide. Wenn Ihre Funktion bereits mit einem Dateisystemtyp konfiguriert ist, müssen Sie ihn entfernen, bevor Sie den anderen konfigurieren.

Konfiguration des Amazon EFS-Dateisystemzugriffs

Sie können eine Funktion konfigurieren, um eine Bindungsbereitstellung für ein Amazon-Elastic-File-System-(Amazon-EFS)-Dateisystem in einem lokalen Verzeichnis zu erstellen. Amazon EFS ist ein serverloses Dateisystem, das sich automatisch an Ihre Workloads anpasst. Mit Amazon EFS kann Ihr Funktionscode auf freigegebene Ressourcen sicher und mit hoher Nebenläufigkeit zugreifen und diese ändern.

Sections

- Ausführungsrolle und Benutzerberechtigungen
- Konfigurieren eines Dateisystems und eines Zugriffspunkts
- Herstellen einer Verbindung mit einem Dateisystem (Konsole)
- Unterstützte Regionen

Ausführungsrolle und Benutzerberechtigungen

Wenn das Dateisystem nicht über eine vom Benutzer konfigurierte Richtlinie AWS Identity and Access Management (IAM) verfügt, verwendet EFS eine Standardrichtlinie, die jedem Client,

der über ein Dateisystem-Mount-Ziel eine Verbindung zum Dateisystem herstellen kann, vollen Zugriff gewährt. Wenn das Dateisystem eine benutzerkonfigurierte IAM-Richtlinie hat, muss die Ausführungsrolle Ihrer Funktion die richtigen `elasticfilesystem`-Berechtigungen besitzen.

Berechtigungen für die Ausführungsrolle

- elastisches Dateisystem: ClientMount
- elastisches Dateisystem: ClientWrite (nicht erforderlich für schreibgeschützte Verbindungen)

Diese Berechtigungen sind in der verwalteten Richtlinie enthalten. AmazonElasticFileSystemClientReadWriteAccess Darüber hinaus muss Ihre Ausführungsrolle die Berechtigungen, die für die Verbindung mit der VPC des Dateisystems erforderlich sind, besitzen.

Wenn Sie ein Dateisystem konfigurieren, werden Ihre Berechtigungen von Lambda verwendet, um Bindungsbereitstellung-Ziele zu überprüfen. Um eine Funktion für die Herstellung von Verbindungen mit einer VPC zu konfigurieren, benötigt Ihr Benutzer die folgenden Berechtigungen:

Benutzerberechtigungen

- elastisches Dateisystem: DescribeMountTargets

Konfigurieren eines Dateisystems und eines Zugriffspunkts

Erstellen Sie ein Dateisystem in Amazon EFS mit einem Bindungsbereitstellung-Ziel in jeder Availability Zone, zu der Ihre Funktion eine Verbindung herstellt. Verwenden Sie für Leistung und Ausfallsicherheit mindestens zwei Availability Zones. In einer einfachen Konfiguration könnten Sie beispielsweise eine VPC mit zwei privaten Subnetzen in separaten Availability Zones haben. Die Funktion verbindet sich mit beiden Subnetzen und ein Mount-Ziel ist in jedem verfügbar. Stellen Sie sicher, dass NFS-Datenverkehr (Port 2049) von den Sicherheitsgruppen erlaubt ist, die von den Funktionen und Mount-Zielen verwendet werden.

> **Note**
>
> Wenn Sie ein Dateisystem erstellen, wählen Sie einen Leistungsmodus, der später nicht geändert werden kann. Der Allzweckmodus hat eine geringere Latenz, und der I/OMax-Modus unterstützt einen höheren maximalen Durchsatz und höhere IOPS. Hilfe bei der Auswahl finden Sie unter Amazon-EFS-Leistung im Amazon-Elastic-File-System-Benutzerhandbuch.

Ein Zugriffspunkt verbindet jede Instance der Funktion mit dem richtigen Mount-Ziel für die Availability Zone, mit der sie eine Verbindung herstellt. Um eine optimale Leistung zu erzielen, erstellen Sie einen Zugriffspunkt mit einem Nicht-Stammpfad und beschränken Sie die Anzahl der Dateien, die Sie in jedem Verzeichnis erstellen. Im folgenden Beispiel wird ein Verzeichnis mit dem Namen `my-function` auf dem Dateisystem erstellt und die Besitzer-ID auf 1001 mit Standardverzeichnisberechtigungen (755) festgelegt.

Example Konfiguration von Zugriffspunkten

- Name (Name – `files`
- Benutzer-ID – `1001`
- Gruppen-ID – `1001`
- Pfad – `/my-function`
- Berechtigungen – `755`
- Benutzer-ID des Eigentümers – `1001`
- Gruppen-Benutzer-ID – `1001`

Wenn eine Funktion den Zugriffspunkt verwendet, erhält sie die Benutzer-ID 1001 und hat vollen Zugriff auf das Verzeichnis.

Weitere Informationen finden Sie in den folgenden Themen im Amazon-Elastic-File-System-Benutzerhandbuch:

- Erstellen von Ressourcen für Amazon EFS
- Arbeiten mit Benutzern, Gruppen und Berechtigungen

Herstellen einer Verbindung mit einem Dateisystem (Konsole)

Eine Funktion stellt eine Verbindung mit einem Dateisystem über das lokale Netzwerk in einer VPC her. Die Subnetze, mit denen Ihre Funktion eine Verbindung herstellt, können dieselben Subnetze sein, die Mounting-Punkte für Ihr Dateisystem enthalten, oder Subnetze in derselben Availability Zone, die NFS-Datenverkehr (Port 2049) an das Dateisystem weiterleiten kann.

> **Note**
>
> Wenn Ihre Funktion noch nicht mit einer VPC verbunden ist, finden Sie weitere Informationen unter Lambda Event Functions Zugriff auf Ressourcen in einer Amazon VPC gewähren.

So konfigurieren Sie den EFS-Dateisystemzugriff

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Konfiguration und dann Dateisysteme.
4. Wählen Sie unter Dateisystem die Option Dateisystem hinzufügen.
5. Wählen Sie EFS aus.
6. Konfigurieren Sie die folgenden Eigenschaften:
   - EFS-Dateisystem – Der Zugriffspunkt für ein Dateisystem in derselben VPC.
   - Lokaler Bindungsbereitstellungspfad – Der Speicherort, an dem das Dateisystem für die Lambda-Funktion gemountet ist, beginnend mit `/mnt/`.

> **Preisgestaltung**
>
> Amazon EFS erhebt Gebühren für Speicher und Durchsatz, mit Raten, die je nach Speicherklasse variieren. Details dazu finden Sie unter Amazon-EFS-Preise.
> Lambda erhebt Gebühren für die Datenübertragung zwischen VPCs. Dies gilt nur, wenn die VPC Ihrer Funktion auf eine andere VPC mit einem Dateisystem gepeered wird. Die Preise sind die gleichen wie für die Amazon-EC2-Datenübertragung zwischen VPCs in derselben Region. Einzelheiten finden Sie unter Lambda – Preise.

Unterstützte Regionen

Amazon EFS for Lambda ist in allen Handelsregionen außer Asien-Pazifik (Neuseeland), Asien-Pazifik (Taipeh), Asien-Pazifik (Malaysia), Asien-Pazifik (Thailand) und Kanada West (Calgary) verfügbar.

Konfiguration des Zugriffs auf Amazon S3 S3-Dateien

Amazon S3 Files bietet ein gemeinsam genutztes Dateisystem, das jede AWS Rechenressource direkt mit Ihren Daten in Amazon S3 verbindet. Amazon S3 Files ermöglicht den Zugriff auf Ihre Amazon S3 S3-Objekte als Dateien mithilfe von standardmäßigen Dateisystemoperationen wie Lesen und Schreiben auf dem lokalen Bereitstellungspfad. Erfahren Sie mehr über Amazon S3 Files.

Sections

- Voraussetzungen und Einrichtung
- Ausführungsrolle und Benutzerberechtigungen
- Herstellen einer Verbindung mit einem Dateisystem (Konsole)
- Direkte Lesevorgänge konfigurieren

Voraussetzungen und Einrichtung

Bevor Sie Amazon S3 Files mit Ihrer Lambda-Funktion einrichten, stellen Sie sicher, dass Sie über Folgendes verfügen:

- Ein Amazon S3 S3-Dateisystem und Mount-Ziele im verfügbaren Status im selben Konto und AWS-Region wie Ihre Lambda-Funktion.
- Eine Lambda-Funktion in derselben VPC wie das Mount-Ziel. In jedem Subnetz, in dem Ihre Funktion bereitgestellt wird, muss ein Mount-Ziel vorhanden sein.
- Sicherheitsgruppen, die NFS-Verkehr (Port 2049) zwischen Ihrer Lambda-Funktion und den Mount-Zielen zulassen. Erfahren Sie mehr über die Konfiguration von Sicherheitsgruppen.

Weitere Informationen finden Sie in den folgenden Themen im Amazon S3 S3-Benutzerhandbuch:

- Erste Schritte mit Amazon S3 Files
- Voraussetzungen Amazon S3 S3-Dateien
- Bewährte Methoden für Amazon S3 S3-Dateien

Ausführungsrolle und Benutzerberechtigungen

Die Ausführungsrolle Ihrer Funktion muss über die folgenden Berechtigungen für den Zugriff auf ein Amazon S3 Files-Dateisystem verfügen:

Berechtigungen für die Ausführungsrolle

- s3files: ClientMount — Erforderlich, um das Dateisystem zu mounten.
- s3files: ClientWrite — Erforderlich für Lese- und Schreibzugriff. Nicht erforderlich für schreibgeschützte Verbindungen.

Diese Berechtigungen sind in der AmazonS3FilesClientReadWriteAccessverwalteten Richtlinie enthalten. Darüber hinaus muss Ihre Ausführungsrolle die Berechtigungen, die für die Verbindung mit der VPC des Dateisystems erforderlich sind, besitzen.

> **Note**
>
> Amazon S3 Files optimiert den Durchsatz, indem berechtigte Leseanfragen direkt aus Ihrem Amazon S3 S3-Bucket gestreamt werden. Direkte Lesevorgänge sind standardmäßig für Funktionen aktiviert, die mit 512 MB oder mehr Arbeitsspeicher konfiguriert sind. Weitere Informationen finden Sie unter Direkte Lesevorgänge konfigurieren.

Ihre Funktion benötigt außerdem die folgenden Berechtigungen, um direkt aus Amazon S3 lesen zu können:

- s3: GetObject
- s3: GetObjectVersion

Weitere Informationen zu den erforderlichen Berechtigungen finden Sie unter IAM-Berechtigungen für Amazon S3 S3-Dateien im Amazon S3 S3-Benutzerhandbuch.

Wenn Sie ein Dateisystem in der Konsole konfigurieren, verwendet Lambda Ihre Berechtigungen, um Mount-Ziele und Access Points zu überprüfen. Um eine Funktion für die Herstellung von Verbindungen mit einer VPC zu konfigurieren, benötigt Ihr Benutzer die folgenden Berechtigungen:

Benutzerberechtigungen

- s3-Dateien: ListFileSystems
- s3-Dateien: ListAccessPoints
- s3-Dateien: GetFileSystem
- s3-Dateien: GetAccessPoint

- s3files: CreateAccessPoint — Wird benötigt, wenn das Dateisystem von der Konsole aus an die Funktion angehängt wird.

Die folgende Beispielrichtlinie gewährt der Ausführungsrolle Ihrer Funktion die Berechtigung, ein Amazon S3-Dateisystem mit Lese-Schreibzugriff zu mounten und direkt aus Amazon S3 zu lesen.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3FilesLambdaAccess",
            "Effect": "Allow",
            "Action": [
                "s3files:ClientMount",
                "s3files:ClientWrite"
            ],
            "Resource": "*"
        },
        {
            "Sid": "S3DirectRead",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": "arn:aws:s3:::bucket-name/*"
        },
        {
            "Sid": "S3FilesConsoleSetup",
            "Effect": "Allow",
            "Action": [
                "s3files:ListFileSystems",
                "s3files:ListAccessPoints",
                "s3files:GetFileSystem",
                "s3files:GetAccessPoint",
                "s3files:CreateAccessPoint"
            ],
            "Resource": "*"
        }
    ]
}
```

Herstellen einer Verbindung mit einem Dateisystem (Konsole)

Eine Funktion stellt eine Verbindung mit einem Dateisystem über das lokale Netzwerk in einer VPC her. Die Subnetze, mit denen Ihre Funktion eine Verbindung herstellt, können dieselben Subnetze sein, die Mounting-Punkte für Ihr Dateisystem enthalten, oder Subnetze in derselben Availability Zone, die NFS-Datenverkehr (Port 2049) an das Dateisystem weiterleiten kann.

> **Note**
>
> Wenn Ihre Funktion noch nicht mit einer VPC verbunden ist, finden Sie weitere Informationen unter Lambda Event Functions Zugriff auf Ressourcen in einer Amazon VPC gewähren.

Um den Zugriff auf S3-Dateien zu konfigurieren

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Konfiguration und dann Dateisysteme.
4. Wählen Sie Dateisystem hinzufügen (oder Bearbeiten, um eine bestehende Konfiguration zu ändern).
5. Wählen Sie S3-Dateien aus.
6. Konfigurieren Sie die folgenden Eigenschaften:
   - S3-Dateisystem — Wählen Sie ein Dateisystem aus der Drop-down-Liste aus.
   - Zugriffspunkt (optional) — Wählen Sie einen Zugriffspunkt. Wenn das Dateisystem keine Zugriffspunkte hat, erstellt Lambda beim Speichern automatisch einen (UID/GID 1000:1000, Stammverzeichnis`/lambda`, Berechtigungen 755). Wenn Access Points vorhanden sind, müssen Sie einen auswählen.
   - Lokaler Bindungsbereitstellungspfad – Der Speicherort, an dem das Dateisystem für die Lambda-Funktion gemountet ist, beginnend mit `/mnt/`.
   - Erweiterte Konfiguration
     - Direkte S3-Lesevorgänge — Steuert, ob Lambda geeignete Lesevorgänge direkt aus Ihrem Amazon S3 S3-Bucket streamen kann, um den Durchsatz zu erhöhen. Weitere Informationen finden Sie unter Direkte Lesevorgänge konfigurieren.
7. Wählen Sie Save (Speichern) aus.

Ihr Dateisystem wird angehängt, wenn Sie Ihre Lambda-Funktion das nächste Mal aufrufen.

Direkte Lesevorgänge konfigurieren

Direkte Lesevorgänge optimieren den Durchsatz, indem sie Daten direkt aus Ihrem Amazon S3 S3-Bucket streamen. Lambda verwendet direkte Lesevorgänge für große Lesevorgänge und wenn der Hochleistungsspeicher des Dateisystems die Dateidaten nicht enthält. Weitere Informationen finden Sie unter So sorgt S3 Files für Leistung. Diese optionale Konfiguration ist Teil der Funktion „`FileSystemConfigs`Für Sie“ und gilt nur für Amazon S3 S3-Dateisysteme. Die `DirectS3Read` Einstellung akzeptiert die folgenden Werte:

- `AUTO`(Standard) — Ermöglicht direkte Lesevorgänge für Funktionen, die mit 512 MB oder mehr Arbeitsspeicher konfiguriert sind.
- `ENABLED`— Ermöglicht direkte Lesevorgänge für Ihre Funktion, einschließlich Funktionen mit weniger als 512 MB Speicher.
- `DISABLED`— Leitet alle Lesevorgänge über den Hochleistungsspeicher des Dateisystems weiter.

> **Note**
>
> Wenn ein direkter Lesevorgang fehlschlägt, greift Lambda automatisch auf das Lesen durch das Dateisystem zurück.

Um direkte Lesevorgänge verwenden zu können, muss die Ausführungsrolle Ihrer Funktion über die erforderlichen Berechtigungen verfügen. Weitere Informationen finden Sie unter Richtlinien für die Voraussetzungen für Amazon S3 Files.

Erstellen Sie einen Alias für eine Lambda-Ereignisfunktion

Sie können für Ihre Lambda-Funktion Aliasse erstellen. Ein Lambda-Alias ist ein Zeiger auf eine Version einer Funktion, der aktualisiert werden kann. Die Benutzer der Funktion können über den Amazon-Ressourcennamen (ARN) des Alias auf die Funktionsversion zugreifen. Wenn Sie eine neue Version bereitstellen, können Sie den Alias aktualisieren, um ihn auf die neue Version umzustellen oder den Datenverkehr zwischen zwei Versionen aufzuteilen. Weitere Hinweise zur Verwendung von Aliasen in Ihren Anwendungen finden Sie unter. the section called “Verwenden von Aliassen”

Console

So erstellen Sie einen Alias mit der Konsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Aliasse und dann Alias erstellen aus.
4. Führen Sie auf der Seite Alias erstellen die folgenden Schritte aus:
   a. Machen Sie für den Alias eine Angabe unter Name.
   b. (Optional) Geben Sie eine Beschreibung für den Alias ein.
   c. Wählen Sie unter Version, eine Funktionsversion aus, auf die der Alias verweisen soll.
   d. (Optional) Um das Routing für den Alias zu konfigurieren, erweitern Sie den Eintrag Gewichteter Alias. Weitere Informationen finden Sie unter Implementieren Sie Lambda Canary-Bereitstellungen mithilfe eines gewichteten Alias.
   e. Wählen Sie Save (Speichern) aus.

AWS CLI

Verwenden Sie den Befehl create-alias, um einen Alias mit AWS Command Line Interface (AWS CLI) zu erstellen.

```
aws lambda create-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number \
  --description " "
```

Verwenden Sie den Befehl update-alias, um einen Alias so zu ändern, dass er auf eine neue Version der Funktion verweist.

```
aws lambda update-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number
```

Verwenden Sie den Befehl delete-alias, um einen Alias zu löschen.

```
aws lambda delete-alias \
  --function-name my-function \
  --name alias-name
```

Die AWS CLI Befehle in den vorherigen Schritten entsprechen den folgenden Lambda-API-Operationen:

- CreateAlias
- UpdateAlias
- DeleteAlias

Verwendung von Lambda-Aliasen in Ereignisquellen und Berechtigungsrichtlinien

Jeder Alias hat eine eindeutige ARN. Ein Alias kann nur auf eine Funktionsversion verweisen, nicht auf einen anderen Alias. Sie können einen Alias aktualisieren, um auf eine neue Version der Funktion zu verweisen.

Ereignisquellen wie Amazon Simple Storage Service (Amazon S3) rufen Ihre Lambda-Funktion auf. Diese Ereignisquellen pflegen eine Zuweisung, die die Funktion identifiziert, die beim Auftreten von Ereignissen aufgerufen werden soll. Wenn Sie in der Mappingkonfiguration einen Lambda-Funktionsalias angeben, müssen Sie das Mapping nicht aktualisieren, wenn sich die Funktionsversion ändert. Weitere Informationen finden Sie unter Wie Lambda Datensätze aus Stream- und warteschlangenbasierten Ereignisquellen verarbeitet.

In einer Ressourcenrichtlinie können Sie Berechtigungen für Ereignisquellen zur Verwendung Ihrer Lambda-Funktion erteilen. Wenn Sie in der Richtlinie einen Alias-ARN angeben, müssen Sie die Richtlinie nicht aktualisieren, wenn sich die Funktionsversion ändert.

Ressourcenrichtlinien

Sie können eine ressourcenbasierte Richtlinie verwenden, um einem Dienst, einer Ressource oder einem Konto Zugriff auf Ihre Funktion zu gewähren. Der Umfang dieser Berechtigung hängt davon ab, ob Sie sie auf einen Alias, eine Version oder die gesamte Funktion anwenden. Wenn Sie beispielsweise einen Aliasnamen (wie`helloworld:PROD`) verwenden, können Sie die `helloworld` Funktion mit dem Alias ARN (`helloworld:PROD`) aufrufen.

Wenn Sie versuchen, die Funktion ohne Alias oder eine bestimmte Version aufzurufen, erhalten Sie einen Berechtigungsfehler. Dieser Berechtigungsfehler tritt weiterhin auf, selbst wenn Sie versuchen, die dem Alias zugeordnete Funktionsversion direkt aufzurufen.

Beispielsweise erteilt der folgende AWS CLI Befehl Amazon S3 die Berechtigung, den PROD-Alias der `helloworld` Funktion aufzurufen, wenn Amazon S3 im Namen von handelt. `amzn-s3-demo-bucket`

```
aws lambda add-permission \
  --function-name helloworld \
  --qualifier PROD \
  --statement-id 1 \
  --principal s3.amazonaws.com \
  --action lambda:InvokeFunction \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
```

```
  --source-account 123456789012
```

Weitere Hinweise zum Verwenden von Ressourcennamen in Richtlinien finden Sie unter Fine-tuning die Abschnitte „Ressourcen“ und „Bedingungen“ der Richtlinien.

Implementieren Sie Lambda Canary-Bereitstellungen mithilfe eines gewichteten Alias

Sie können einen gewichteten Alias verwenden, um den Datenverkehr auf zwei verschiedene Versionen derselben Funktion aufzuteilen. Mit diesem Ansatz können Sie neue Versionen Ihrer Funktionen mit einem kleinen Prozentsatz des Datenverkehrs testen und bei Bedarf schnell wieder zurücksetzen. Dies wird als Canary-Bereitstellung bezeichnet. Kanarische Bereitstellungen unterscheiden sich von blue/green Bereitstellungen dadurch, dass die neue Version nur einem Teil der Anfragen ausgesetzt wird, anstatt den gesamten Datenverkehr auf einmal umzuleiten.

Sie können ein Alias auf maximal zwei Lambda-Funktionsversionen verweisen lassen. Die Versionen müssen folgende Kriterien erfüllen:

- Beide Versionen müssen die gleiche Ausführungsrolle haben.
- Beide Versionen müssen die gleiche Warteschlangenkonfiguration für unzustellbare Nachrichten oder keine Warteschlangenkonfiguration für unzustellbare Nachrichten haben.
- Beide Versionen müssen veröffentlicht werden. Der Alias darf nicht auf verweise `$LATEST`.

> **Note**
>
> Lambda verwendet ein einfaches probabilistisches Modell, um den Datenverkehr zwischen den beiden Funktionsversionen zu verteilen. Bei niedrigem Datenverkehr sehen Sie möglicherweise eine hohe Abweichung zwischen dem konfigurierten und dem tatsächlichen Prozentsatz des Datenverkehrs für jede Version. Wenn Ihre Funktion bereitgestellte Parallelität verwendet, können Sie Überlaufaufrufe durch Konfigurieren einer höheren Anzahl von bereitgestellten Parallelitätsinstances während des aktiven Alias-Routings vermeiden.

Einen gewichteten Alias erstellen

Console

So konfigurieren Sie das Routing für einen Alias über die Konsole

> **Note**
>
> Stellen Sie sicher, dass die Funktion über mindestens zwei veröffentlichte Versionen verfügt. Um weitere Versionen zu erstellen, folgen Sie den Anweisungen unter Erstellen von Funktionsversionen.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Aliasse und dann Alias erstellen aus.
4. Führen Sie auf der Seite Alias erstellen die folgenden Schritte aus:
   a. Machen Sie für den Alias eine Angabe unter Name.
   b. (Optional) Geben Sie eine Beschreibung für den Alias ein.
   c. Wählen Sie unter Version die erste Funktionsversion aus, auf die der Alias verweisen soll.
   d. Erweitern Sie Gewichteter Alias.
   e. Wählen Sie unter Zusätzliche Version die zweite Funktionsversion aus, auf die der Alias verweisen soll.
   f. Geben Sie unter Gewicht (%) einen Gewichtungswert für die Funktion ein. Die Gewichtung ist der Prozentsatz des Datenverkehrs, der beim Aufruf des Alias dieser Version zugewiesen wird. Die erste Version erhält den Datenverkehr mit der verbleibenden Gewichtung. Wenn Sie beispielsweise 10 Prozent für die Additional version (Zusätzliche Version) angeben, werden der ersten Version automatisch 90 Prozent zugewiesen.
   g. Wählen Sie Speichern.

AWS CLI

Verwenden Sie die AWS CLI Befehle create-alias und update-alias, um die Datenverkehrsgewichte zwischen zwei Funktionsversionen zu konfigurieren. Wenn Sie den Alias erstellen oder aktualisieren, geben Sie die Datenverkehrsgewichtung im `routing-config`-Parameter an.

Im folgenden Beispiel wird ein Lambda-Funktionsalias namens routing-alias erstellt, der auf Version 1 der Funktion verweist. Version 2 der Funktion erhält 3 Prozent des Datenverkehrs. Die verbleibenden 97 Prozent des Datenverkehrs werden zu Version 1 weitergeleitet.

```
aws lambda create-alias \
  --name routing-alias \
  --function-name my-function \
  --function-version 1  \
  --routing-config AdditionalVersionWeights={"2"=0.03}
```

Verwenden Sie den `update-alias`-Befehl, um den Prozentsatz des eingehenden Datenverkehrs an Version 2 zu erhöhen. Im folgenden Beispiel erhöhen Sie den Datenverkehr auf 5 Prozent.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function \
  --routing-config AdditionalVersionWeights={"2"=0.05}
```

Um den gesamten Datenverkehr an Version 2 weiterzuleiten, ändern Sie den Befehl `update-alias`, um die Eigenschaft `function-version` so zu ändern, dass der Alias auf Version 2 verweist. Der Befehl setzt auch die Weiterleitungskonfiguration zurück.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function  \
  --function-version 2 \
  --routing-config AdditionalVersionWeights={}
```

Die AWS CLI Befehle in den vorherigen Schritten entsprechen den folgenden Lambda-API-Operationen:

- CreateAlias

- UpdateAlias

Feststellen, welche Version aufgerufen wurde

Wenn Sie Datenverkehrsgewichtungen zwischen zwei Funktionsversionen konfigurieren, gibt es zwei Möglichkeiten, die aufgerufene Lambda-Funktionsversion zu bestimmen:

- CloudWatch Logs — Lambda gibt automatisch einen `START` Protokolleintrag aus, der die aufgerufene Versions-ID für jeden Funktionsaufruf enthält. Beispiel:

  ```
  START RequestId: 1dh194d3759ed-4v8b-a7b4-1e541f60235f Version: 2
  ```

  Für Aliasaufrufe verwendet Lambda die Dimension `ExecutedVersion` zum Filtern der Metrikdaten nach aufgerufener Version. Weitere Informationen finden Sie unter Metriken für Lambda-Ereignisfunktionen anzeigen.
- Antwortnutzlast (synchrone Aufrufe) – Antworten auf synchrone Funktionsaufrufe enthalten einen `x-amz-executed-version`-Header, der angibt, welche Funktionsversion aufgerufen wurde.

Erstellen Sie eine fortlaufende Bereitstellung mit gewichteten Aliasnamen

Verwenden Sie AWS CodeDeploy und AWS Serverless Application Model (AWS SAM), um eine fortlaufende Bereitstellung zu erstellen, die automatisch Änderungen an Ihrem Funktionscode erkennt, eine neue Version Ihrer Funktion bereitstellt und die Menge des Datenverkehrs, der zur neuen Version fließt, schrittweise erhöht. Der Datenverkehr und die Erhöhungsrate sind Parameter, die Sie konfigurieren können.

AWS SAM Führt in einer fortlaufenden Bereitstellung die folgenden Aufgaben aus:

- Konfiguriert Ihre Lambda-Funktion und erstellt einen Alias. Die gewichtete Alias-Weiterleitungskonfiguration ist die zugrunde liegende Funktion, die die fortlaufende Bereitstellung implementiert.
- Erstellt eine CodeDeploy Anwendung und eine Bereitstellungsgruppe. Die Bereitstellungsgruppe verwaltet die fortlaufende Bereitstellung und das Rollback, falls erforderlich.
- Erkennt, wenn Sie eine neue Version Ihrer Lambda-Funktion erstellen.
- Löst aus CodeDeploy , um die Bereitstellung der neuen Version zu starten.

Beispiel AWS SAM Vorlage

Das folgende Beispiel zeigt eine AWS SAM -Vorlage für eine einfache fortlaufende Bereitstellung.

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions

Resources:
# Details about the myDateTimeFunction Lambda function
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: myDateTimeFunction.handler
      Runtime: nodejs24.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

Diese Vorlage definiert eine Lambda-Funktion mit der Bezeichnung `myDateTimeFunction` und den folgenden Eigenschaften.

AutoPublishAlias

Die `AutoPublishAlias`-Eigenschaft erstellt einen Alias mit der Bezeichnung `live`. Darüber hinaus erkennt das AWS SAM -Framework automatisch, wenn Sie neuen Code für die Funktion speichern. Das Framework veröffentlicht dann eine neue Funktionsversion und aktualisiert den `live`-Alias so, dass er auf die neue Version verweist.

DeploymentPreference

Die `DeploymentPreference` Eigenschaft bestimmt die Geschwindigkeit, mit der die CodeDeploy Anwendung den Datenverkehr von der ursprünglichen Version der Lambda-Funktion auf die neue Version verlagert. Der Wert `Linear10PercentEvery2Minutes` verschiebt alle 2 Minuten weitere 10 Prozent des Datenverkehrs auf die neue Version.

Eine Liste der vordefinierten Bereitstellungskonfigurationen finden Sie unter Bereitstellungskonfigurationen.

Weitere Informationen zum Erstellen fortlaufender Bereitstellungen mit CodeDeploy und AWS SAM finden Sie im Folgenden:

- Tutorial: Stellen Sie eine aktualisierte Lambda-Funktion bereit mit CodeDeploy und dem AWS Serverless Application Model
- Schrittweise Bereitstellung serverloser Anwendungen mit AWS SAM

Versionen der Lambda-Ereignisfunktion verwalten

Sie können Versionen verwenden, um die Bereitstellung Ihrer -Funktionen zu verwalten. Beispielsweise können Sie eine neue Version einer Funktion für Beta-Tests veröffentlichen, ohne dass Benutzer der stabilen Produktionsversion betroffen sind. Lambda erstellt jedes Mal, wenn Sie die Funktion veröffentlichen, eine neue Version Ihrer Funktion. Die neue Version ist eine Kopie der unveröffentlichten Version der Funktion. Die unveröffentlichte Version heißt `$LATEST`.

Wichtig ist, dass Sie bei jeder Bereitstellung Ihres Funktionscodes den aktuellen Code in `$LATEST` überschreiben. Um die aktuelle Iteration von `$LATEST` zu speichern, erstellen Sie eine neue Funktionsversion. Wenn `$LATEST` mit einer zuvor veröffentlichten Version identisch ist, können Sie erst dann eine neue Version erstellen, wenn Sie Änderungen an `$LATEST` bereitgestellt haben. Zu diesen Änderungen können die Aktualisierung des Codes oder die Änderung der Funktionskonfigurationseinstellungen gehören.

Nachdem Sie eine Funktionsversion veröffentlicht haben, sind Code, Laufzeit, Architektur, Speicher, Ebenen und die meisten anderen Konfigurationseinstellungen unveränderlich. Das bedeutet, dass Sie diese Einstellungen nicht ändern können, ohne eine neue Version von `$LATEST` zu veröffentlichen. Sie können die folgenden Elemente für eine veröffentlichte Funktionsversion konfigurieren:

- Auslöser
- Ziele
- Bereitgestellte Gleichzeitigkeit
- Asynchroner Aufruf
- Datenbankverbindungen und Proxys

> **Note**
>
> Bei Verwendung von Steuerelementen zur Laufzeitverwaltung im Modus Auto (Automatisch) wird die Laufzeitversion, die von der Funktionsversion verwendet wird, automatisch aktualisiert. Bei Verwendung des Modus Function update (Funktionsaktualisierung) oder Manual (Manuell) wird die Laufzeitversion nicht aktualisiert. Weitere Informationen finden Sie unter the section called “Aktualisierungen der Laufzeitversion”.

Sections

- Erstellen von Funktionsversionen
- Verwenden von Versionen
- Gewähren von Berechtigungen

Erstellen von Funktionsversionen

Sie können den Funktionscode und die Einstellungen nur in der nicht veröffentlichten Version einer Funktion ändern. Wenn Sie eine Version veröffentlichen, sperrt Lambda den Code und die meisten Einstellungen, um eine konsistente Erfahrung für Benutzer dieser Version zu gewährleisten.

Sie können eine Funktionsversion mit der Lambda-Konsole erstellen.

So erstellen Sie eine neue Funktionsversion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus und dann den Tab Versionen.
3. Wählen Sie auf der Versionskonfigurationsseite die Option Neue Version veröffentlichen aus.
4. (Optional) Geben Sie eine Versionsbeschreibung ein.
5. Wählen Sie Publish.

Alternativ können Sie mithilfe der PublishVersionAPI-Operation eine Version einer Funktion veröffentlichen.

Der folgende AWS CLI Befehl veröffentlicht eine neue Version einer Funktion. In der Antwort werden die Konfigurationsinformationen der neuen Version zurückgegeben, einschließlich der Versionsnummer und des Funktion-ARN mit dem Versionssuffix.

```
aws lambda publish-version --function-name my-function
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
```

```
  ...
}
```

> **Note**
>
> Lambda weist monoton steigende Sequenznummern für die Versionsverwaltung zu. Lambda verwendet niemals Versionsnummern wieder, auch nicht nachdem Sie eine Funktion gelöscht und neu erstellt haben.

Verwenden von Versionen

Sie können Ihre Lambda-Funktion entweder mit einem qualifizierten ARN oder einem nicht qualifizierten ARN referenzieren.

- Qualifizierter ARN – Hierbei handelt es sich um den ARN der Funktion mit einem Versionssuffix. Das folgende Beispiel bezieht sich auf Version 42 der Funktion `helloworld`.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld:42
  ```

- Unqualifizierter ARN – Hierbei handelt es sich um den ARN der Funktion ohne ein Versionssuffix.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld
  ```

Sie können in allen relevanten API-Operationen einen qualifizierten oder einen nicht qualifizierten ARN verwenden. Sie können jedoch keinen unqualifizierten ARN verwenden, um einen Alias zu erstellen.

Wenn Sie sich entscheiden, Funktionsversionen nicht zu veröffentlichen, können Sie entweder den qualifizierten oder den nicht qualifizierten ARN in Ihrer Ereignisquellen-Zuweisung verwenden, um die Funktion aufzurufen. Wenn Sie eine Funktion mit einem unqualifizierten ARN aufrufen, ruft Lambda implizit `$LATEST` auf.

Der qualifizierte ARN für jede Lambda-Funktionsversion ist eindeutig. Nachdem Sie eine Version veröffentlicht haben, können Sie den ARN oder den Funktionscode nicht ändern.

Lambda veröffentlicht eine neue Funktionsversion nur dann, wenn der Code noch nie veröffentlicht wurde oder wenn sich der Code im Vergleich zur zuletzt veröffentlichten Version geändert hat. Wenn keine Änderung vorliegt, bleibt die Funktionsversion die zuletzt veröffentlichte Version.

Wenn Sie eine Version veröffentlichen, erstellt Lambda einen unveränderlichen Snapshot des Codes und der Konfiguration Ihrer Funktion. Nicht alle Konfigurationsänderungen lösen die Veröffentlichung einer neuen Version aus. Die folgenden Konfigurationsänderungen qualifizieren eine Funktion für die Versionsveröffentlichung:

- Funktionscode
- Umgebungsvariablen
- Laufzeit
- Handler
- Layer
- Arbeitsspeichergröße
- Zeitüberschreitung
- VPC-Konfiguration
- Konfiguration der Warteschlange für unzustellbare Nachrichten
- IAM role (IAM-Rolle)
- Description
- Architektur (x86_64 oder arm64)
- Größe des flüchtigen Speichers
- Art Package
- Codespeichermodus (Umschalten zwischen Lambda-managed und selbstverwaltetem S3-Speicher)
- Protokollierungskonfiguration
- Dateisystemkonfiguration
- SnapStart
- Nachverfolgungskonfiguration

Betriebseinstellungen wie reservierte Parallelität lösen nicht die Veröffentlichung einer neuen Version aus, wenn sie geändert werden.

Gewähren von Berechtigungen

Sie können eine ressourcenbasierte Richtlinie oder eine identitätsbasierte Richtlinie verwenden, um Zugriff auf Ihre Funktion zu gewähren. Der Umfang der Berechtigung hängt davon ab, ob Sie die

Richtlinie auf eine Funktion oder auf eine Version einer Funktion anwenden. Weitere Hinweise zu Funktionsressourcennamen in Richtlinien finden Sie unter Fine-tuning die Abschnitte „Ressourcen“ und „Bedingungen“ der Richtlinien.

Sie können die Verwaltung von Ereignisquellen und AWS Identity and Access Management (IAM-) Richtlinien vereinfachen, indem Sie Funktionsaliase verwenden. Weitere Informationen finden Sie unter Erstellen Sie einen Alias für eine Lambda-Ereignisfunktion.

Verwenden von Tags für Lambda-Ereignisfunktionen

Sie können Tag-Funktionen markieren, um Ihre Ressourcen zu organisieren und zu verwalten. Tags sind frei formulierte Schlüssel-Wert-Paare, die mit Ihren Ressourcen verknüpft sind und von AWS-Services unterstützt werden. Weitere Informationen zu Anwendungsfällen für Tags finden Sie unter Allgemeine Tagging-Strategien im Tagging Resources and Tag AWS Editor Guide.

Tags gelten für die Funktionsebene, nicht für Versionen oder Aliase. Tags sind nicht Teil der versionsspezifischen Konfiguration, die einen Snapshot davon AWS Lambda erstellt, wann Sie eine Version veröffentlichen. Sie können die Lambda-API verwenden, um Tags anzuzeigen und zu aktualisieren. Sie können Tags auch anzeigen und aktualisieren, während Sie eine bestimmte Funktion in der Lambda-Konsole verwalten.

Sie können Tags auch verwenden, um den Zugriff auf Ihre Funktionen mit der attributebasierten Zugriffskontrolle (ABAC) zu steuern. Weitere Informationen finden Sie unter the section called “Attribute-based Zugriffskontrolle”.

Sections

- Erforderliche Berechtigungen zum Arbeiten mit Tags
- Verwendung von Tags mit der Lambda-Konsole
- Verwenden von Tags mit AWS CLI

Erforderliche Berechtigungen zum Arbeiten mit Tags

Um einer AWS Identity and Access Management -(IAM)-Identität (Benutzer, Gruppe oder Rolle) das Lesen oder Setzen von Tags auf einer Ressource zu ermöglichen, gewähren Sie ihr die entsprechenden Berechtigungen:

- lambda: ListTags —Wenn eine Ressource Tags enthält, gewähren Sie diese Berechtigung jedem, der sie aufrufen muss. `ListTags` Für getaggte Event-Funktionen ist diese Berechtigung auch für erforderlich. `GetFunction`
- lambda: TagResource —Erteilen Sie diese Berechtigung jedem, der bei create aufrufen `TagResource` oder ein Tag ausführen muss.

Optional können Sie erwägen, auch die lambda: UntagResource -Berechtigung zu erteilen, um `UntagResource` Aufrufe der Ressource zuzulassen.

Weitere Informationen finden Sie unter Identity-based IAM-Richtlinien für Lambda.

Verwendung von Tags mit der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um Funktionen mit Tags zu erstellen, Tags zu vorhandenen Funktionen hinzuzufügen und Funktionen nach von Ihnen hinzugefügten Tags zu filtern.

Hinzufügen von Tags beim Erstellen einer Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Klicken Sie auf Ohne Vorgabe erstellen oder Container-Image.
4. Richten Sie unter Grundinformationen Ihre Funktion ein. Weitere Hinweise zum Konfigurieren von Funktionen finden Sie unter the section called “Konfiguration von Ereignisfunktionen”.
5. Erweitern Sie Advanced settings (Erweiterte Einstellungen) und wählen Sie dann Enable tags (Tags aktivieren) aus.
6. Wählen Sie Add new tag (Neues Tag hinzufügen) und geben Sie dann einen Key (Schlüssel) und einen optionalen Value (Wert) ein. Wiederholen Sie diesen Schritt, um weitere Tags hinzuzufügen.
7. Wählen Sie „Event-Funktion erstellen“.

Hinzufügen von Tags zu einer bestehenden Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuraton (Konfiguration) und dann Tags aus.
4. Wählen Sie unter Tags die Option Manage tags (Tags verwalten) aus.
5. Wählen Sie Add new tag (Neues Tag hinzufügen) und geben Sie dann einen Key (Schlüssel) und einen optionalen Value (Wert) ein. Wiederholen Sie diesen Schritt, um weitere Tags hinzuzufügen.
6. Wählen Sie Speichern.

So filtern Sie Funktionen mit Tags

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Wählen Sie die Suchleiste aus, um eine Liste mit Funktionseigenschaften und Tag-Schlüsseln anzuzeigen.
3. Wählen Sie einen Tag-Schlüssel, um eine Liste der Werte anzuzeigen, die in der aktuellen AWS Region verwendet werden.
4. Wählen Sie Verwenden: „Tag-Name“, um alle Funktionen zu sehen, die mit dieser Taste gekennzeichnet sind oder wählen Sie einen Operator, um weiter nach Werten zu filtern.
5. Wählen Sie Ihren Tag-Wert aus, um nach einer Kombination aus Tag-Schlüssel und -Wert zu filtern.

Die Suchleiste unterstützt auch die Suche nach Tag-Schlüsseln. Geben Sie `tag` ein, um nur eine Liste der Tag-Schlüssel anzuzeigen, oder geben Sie den Namen eines Schlüssels ein, um ihn in der Liste zu suchen.

Verwenden von Tags mit AWS CLI

Mit der Lambda-API können Sie Tags zu vorhandenen Lambda-Ressourcen, einschließlich Funktionen, hinzufügen und entfernen. Sie können bei der Erstellung einer Funktion auch Tags hinzufügen, um eine Ressource während ihres gesamten Lebenszyklus mit Tags zu versehen.

Aktualisierung von Tags mit den Lambda-Tag-APIs

Sie können Tags für unterstützte Lambda-Ressourcen über die TagResourceund UntagResourceAPI-Operationen hinzufügen und entfernen.

Sie können diese Operationen mit der Taste AWS CLI aufrufen. Um Tags einer vorhandenen Ressource hinzuzufügen, verwenden Sie den folgenden `tag-resource`-Befehl. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Mit dem Befehl `untag-resource` können Sie Tags entfernen. In diesem Beispiel wird das Tag mit dem Schlüssel entfernt*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
```

```
--tag-keys Department
```

Hinzufügen von Tags beim Erstellen einer Funktion

Verwenden Sie die CreateFunctionAPI-Operation, um eine neue Lambda-Funktion mit Tags zu erstellen. Geben Sie den Parameter `Tags` an. Sie können diesen Vorgang mit dem `create-function`-CLI-Befehl und der Option --tags aufrufen.

Vergewissern Sie sich vor der Verwendung des Tags-Parameters mit `CreateFunction`, dass Ihre Rolle neben den üblichen Berechtigungen, die für diesen Vorgang erforderlich sind, auch die Berechtigung hat, Ressourcen zu taggen. Weitere Informationen zu den Berechtigungen für das Tagging finden Sie unter the section called “Erforderliche Berechtigungen zum Arbeiten mit Tags”. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *Department* und eines mit dem Schlüssel*CostCenter*.

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs24.x \
--role arn:aws:iam::123456789012:role/lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

Anzeigen von Tags in einer Funktion

Um die Tags anzuzeigen, die auf eine bestimmte Lambda-Ressource angewendet werden, verwenden Sie die `ListTags`-API-Operation. Weitere Informationen finden Sie unter ListTags.

Sie können diesen Vorgang mit dem `list-tags` AWS CLI Befehl aufrufen, indem Sie einen ARN (Amazon Resource Name) angeben.

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier
```

Sie können die Tags anzeigen, die mit der GetFunctionAPI-Operation auf eine bestimmte Ressource angewendet werden. Eine vergleichbare Funktion ist nicht für andere Ressourcentypen verfügbar.

Sie können diesen Vorgang mit dem `get-function`-CLI-Befehl aufrufen:

```
aws lambda get-function --function-name my-function
```

Filtern von Ressourcen nach Tag

Sie können den AWS Resource Groups Tagging API GetResourcesAPI-Vorgang verwenden, um Ihre Ressourcen nach Tags zu filtern. Die `GetResources`-Operation empfängt bis zu 10 Filter, wobei jeder Filter einen Tag-Schlüssel und bis zu 10 Tag-Werte enthält. Sie stellen `GetResources` mit einem `ResourceType` zur Verfügung, um nach bestimmten Ressourcentypen zu filtern.

Sie können diesen Vorgang mit dem `get-resources` AWS CLI Befehl aufrufen. Beispiele für die Verwendung von `get-resources` finden Sie unter get-resources in der AWS -CLI-Befehlsreferenz.

Antwortstreaming für Lambda-Ereignisfunktionen

Lambda-Funktionen können Antwortnutzlasten nativ über Lambda-Funktions-URLs oder mithilfe der InvokeWithResponseStreamAPI (über das AWS SDK oder direkte API-Aufrufe) an Clients zurückstreamen. Ihre Lambda-Funktion kann auch Antwort-Payloads über die Amazon API Gateway Gateway-Proxyintegration streamen, die die InvokeWithResponseStreamAPI verwendet, um Ihre Funktion aufzurufen. Antwort-Streaming kann für latenzempfindliche Anwendungen von Vorteil sein, da es die Leistung in der Zeit bis zum ersten Byte (TTFB) verbessert. Dies liegt daran, dass Sie Teilantworten an den Client zurücksenden können, sobald sie verfügbar sind. Darüber hinaus können Antwort-Streaming-Funktionen Nutzdaten bis zu 200 MB zurückgeben, verglichen mit dem Maximum von 6 MB für gepufferte Antworten.

Das Streamen einer Antwort bedeutet auch, dass Ihre Funktion nicht die gesamte Antwort im Speicher speichern muss. Bei sehr großen Antworten kann dies die Menge des Speichers reduzieren, die Sie für Ihre Funktion konfigurieren müssen. Informationen zur Auswahl zwischen Funktions-URLs und API Gateway für Ihren Anwendungsfall finden Sie unterthe section called “Funktions-URLs im Vergleich zu Amazon API Gateway”.

> **Note**
>
> Lambda-Response-Streaming ist noch nicht in allen AWS-Regionen Bereichen verfügbar. Informationen zur Verfügbarkeit von Funktionen nach Regionen finden Sie unter AWS Funktionen nach Regionen im Builder Center.

Die Geschwindigkeit, mit der Lambda Ihre Antworten streamt, hängt von der Größe der Antwort ab. Die Streaming-Rate für die ersten 6 MB der Antwort Ihrer Funktion ist unbegrenzt. Bei Antworten, die größer als 6 MB sind, gilt für den Rest der Antwort eine Bandbreitenbegrenzung. Weitere Informationen zur Streaming-Bandbreite finden Sie unter Bandbreitenbegrenzung für Antwort-Streaming.

Streaming-Antworten sind mit Kosten verbunden, und gestreamte Antworten werden nicht unterbrochen oder gestoppt, wenn die Verbindung zum aufrufenden Client unterbrochen wird. Kunden wird die volle Funktionsdauer in Rechnung gestellt, weshalb Kunden bei der Konfiguration langer Funktions-Timeouts Vorsicht walten lassen sollten.

Lambda unterstützt Antwortstreaming auf Node.js verwalteten Laufzeiten. Für andere Sprachen, einschließlich Python, können Sie eine benutzerdefinierte Runtime mit einer benutzerdefinierten

Runtime-API-Integration verwenden, um Antworten zu streamen, oder den Lambda Web Adapter verwenden.

> **Note**
>
> Wenn Sie Ihre Funktion über die Lambda-Konsole testen, werden die Antworten immer gepuffert angezeigt.

Informationen zum Schreiben von Streaming-Funktionen finden Sie unterthe section called "Schreiben von Funktionen". Informationen zum Aufrufen von Streaming-Funktionen über Funktions-URLs finden Sie unterthe section called "Aufrufen von -Funktionen". Eine Schritt-für-Schritt-Anleitung finden Sie unter the section called "Tutorial: Erstellen einer Funktion zum Streamen von Antworten mit einer Funktions-URL".

Themen

- Bandbreitenbegrenzung für Antwort-Streaming
- VPC-Kompatibilität mit Antwort-Streaming
- Schreiben von Antwort-Streaming-fähigen Lambda-Funktionen
- Aufrufen einer Antwort-Streaming-fähigen Funktion mit Lambda-Funktions-URLs
- Tutorial: Erstellen einer Lambda-Funktion zum Streamen von Antworten mit einer Funktions-URL

Bandbreitenbegrenzung für Antwort-Streaming

Die ersten 6 MB der Antwort-Payload Ihrer Funktion haben eine unbegrenzte Bandbreite. Nach diesem ersten Burst streamt Lambda Ihre Antwort mit einer maximalen Rate von 2 Mbit/s. Wenn Ihre Funktionsantworten nie größer als 6 MB sind, gilt diese Bandbreitenbegrenzung nicht.

> **Note**
>
> Bandbreitenbeschränkungen gelten nur für die Antwort-Payload Ihrer Funktion und nicht für den Netzwerkzugriff durch Ihre Funktion.

Die Geschwindigkeit der unbegrenzten Bandbreite hängt von einer Reihe von Faktoren ab, einschließlich der Verarbeitungsgeschwindigkeit Ihrer Funktion. Normalerweise können Sie für die ersten 6 MB der Antwort Ihrer Funktion mit einer Geschwindigkeit von mehr als 2 MB/s rechnen.

Wenn Ihre Funktion eine Antwort an ein Ziel außerhalb von streamt AWS, hängt die Streaming-Rate auch von der Geschwindigkeit der externen Internetverbindung ab.

VPC-Kompatibilität mit Antwort-Streaming

Bei der Verwendung von Lambda-Funktionen in einer VPC-Umgebung gibt es wichtige Überlegungen zum Antwort-Streaming:

- Lambda-Funktions-URLs unterstützen kein Antwort-Streaming innerhalb einer VPC-Umgebung.
- Sie können Antwort-Streaming innerhalb einer VPC verwenden, indem Sie Ihre Lambda-Funktion über das AWS SDK mithilfe der API aufrufen. `InvokeWithResponseStream` Das erfordert die Einrichtung der entsprechenden VPC-Endpunkte für Lambda.
- Für VPC-Umgebungen müssen Sie einen VPC-Schnittstellen-Endpunkt für Lambda erstellen, um die Kommunikation zwischen Ihren Ressourcen in der VPC und dem Lambda-Service zu ermöglichen.

Eine typische Architektur für das Antwort-Streaming in einer VPC könnte Folgendes beinhalten:

```
Client in VPC -> Interface VPC endpoint for Lambda -> Lambda function -> Response
 streaming back through the same path
```

Schreiben von Antwort-Streaming-fähigen Lambda-Funktionen

Das Schreiben des Handlers für Antwort-Streaming-Funktionen unterscheidet sich von typischen Handler-Mustern. Wenn Sie Streaming-Funktionen schreiben, sollten Sie Folgendes beachten:

- Umschließen Sie Ihre Funktion mit dem `awslambda.streamifyResponse()`-Decorator. Das globale `awslambda`-Objekt wird von der Lambda-Laufzeitumgebung Node.js bereitgestellt.
- Beenden Sie den Stream ordnungsgemäß, um sicherzustellen, dass die Datenverarbeitung abgeschlossen ist.

Konfigurieren einer Handler-Funktion zum Streamen von Antworten

Um der Laufzeit zu signalisieren, dass Lambda die Antworten Ihrer Funktion streamen soll, müssen Sie Ihre Funktion mit dem `streamifyResponse()`-Decorator wrappen. Dadurch wird die Laufzeit angewiesen, den richtigen Logikpfad für Streaming-Antworten zu verwenden, und ermöglicht es der Funktion, Antworten zu streamen.

Der `streamifyResponse()`-Decorator akzeptiert eine Funktion, die die folgenden Parameter akzeptiert:

- `event` – Stellt Informationen über das Aufrufereignis der Funktions-URL bereit, z. B. die HTTP-Methode, Abfrageparameter und den Anforderungskörper.
- `responseStream` – Stellt einen schreibbaren Stream bereit.
- `context` – Stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Das `responseStream`-Objekt ist ein Node.js `writableStream`. Wie bei jedem solchen Stream sollten Sie die `pipeline()`-Methode verwenden.

> **Note**
>
> Das globale `awslambda`-Objekt wird automatisch von der Lambda-Laufzeit Node.js bereitgestellt, und es ist kein Import erforderlich.

Example Handler mit aktiviertem Antwort-Streaming

```
import { pipeline } from 'node:stream/promises';
import { Readable } from 'node:stream';

export const echo = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

Obwohl `responseStream` die `write()`-Methode bietet, in den Stream zu schreiben, empfehlen wir Ihnen, wo immer möglich, `pipeline()` zu verwenden. Die Verwendung von `pipeline()` stellt sicher, dass der schreibbare Stream nicht von einem schneller lesbaren Stream überfordert wird.

Den Stream beenden

Stellen Sie sicher, dass Sie den Stream ordnungsgemäß beenden, bevor der Handler zurückkehrt. Die `pipeline()`-Methode verarbeitet dies automatisch.

Für andere Anwendungsfälle rufen Sie die `responseStream.end()`-Methode auf, um einen Stream ordnungsgemäß zu beenden. Diese Methode signalisiert, dass keine Daten mehr in den Stream geschrieben werden sollen. Diese Methode ist nicht erforderlich, wenn Sie mit `pipeline()` oder `pipe()` in den Stream schreiben.

Ab Node.js 24 wartet Lambda nicht mehr darauf, dass ungelöste Versprechen abgeschlossen werden, nachdem Ihr Handler zurückgekehrt ist oder der Antwortstream beendet wurde. Wenn Ihre Funktion von zusätzlichen asynchronen Vorgängen wie Timern oder Abrufen abhängt, sollten Sie sie in Ihrem Handler verwenden. `await`

Example Beispiel für das Beenden eines Streams mit Pipeline ()

```
import { pipeline } from 'node:stream/promises';

export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  await pipeline(requestStream, responseStream);
});
```

Example Beispiel für das Beenden eines Streams ohne Pipeline ()

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

Aufrufen einer Antwort-Streaming-fähigen Funktion mit Lambda-Funktions-URLs

> **Note**
>
> Ihre Lambda-Funktion kann jetzt Antwort-Payloads über die Amazon API Gateway Gateway-Proxyintegration streamen.

Sie können Funktionen mit aktiviertem Antwort-Streaming aufrufen, indem Sie den Aufrufmodus der URL Ihrer Funktion ändern. Der Aufrufmodus bestimmt, welche API-Operation Lambda verwendet, um Ihre Funktion aufzurufen. Die verfügbaren Aufrufmodi sind:

- `BUFFERED` – Dies ist die Standardoption. Lambda ruft Ihre Funktion mithilfe der `Invoke`-API-Operation auf. Die Aufrufergebnisse sind verfügbar, wenn die Nutzlast abgeschlossen ist. Die maximale Nutzlastgröße beträgt 6 MB.
- `RESPONSE_STREAM` – Ermöglicht es Ihrer Funktion, die Ergebnisse der Nutzlasten zu streamen, sobald sie verfügbar sind. Lambda ruft Ihre Funktion mithilfe der `InvokeWithResponseStream`-API-Operation auf. Die maximale Größe der Antwort-Payload beträgt 200 MB.

Sie können Ihre Funktion auch ohne Antwort-Streaming aufrufen, indem Sie die `Invoke`-API-Operation direkt aufrufen. Lambda streamt jedoch alle Antwort-Nutzlasten für Aufrufe, die über die URL der Funktion erfolgen, bis Sie den Aufrufmodus auf `BUFFERED` ändern.

Console

Um den Aufrufmodus einer Funktions-URL (Konsole) festzulegen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Funktion, für die Sie den Aufrufmodus festlegen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
4. Wählen Sie Bearbeiten und dann Zusätzliche Einstellungen.
5. Wählen Sie unter Aufrufmodus den gewünschten Aufrufmodus aus.
6. Wählen Sie Speichern.

AWS CLI

Um den Aufrufmodus einer Funktions-URL (AWS CLI) festzulegen

```
aws lambda update-function-url-config \
  --function-name my-function \
  --invoke-mode RESPONSE_STREAM
```

CloudFormation

Um den Aufrufmodus einer Funktions-URL (CloudFormation) festzulegen

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

Weitere Hinweise zum Konfigurieren von Funktions-URLs finden Sie unter Lambda-Funktions-URLs.

Tutorial: Erstellen einer Lambda-Funktion zum Streamen von Antworten mit einer Funktions-URL

In diesem Tutorial erstellen Sie eine Lambda-Funktion, die als ZIP-Dateiarchiv mit einem Funktions-URL-Endpunkt definiert ist, der einen Antwort-Stream zurückgibt. Weitere Hinweise zum Konfigurieren von Funktions-URLs finden Sie unter Funktions-URLs.

Voraussetzungen

In diesem Tutorial wird davon ausgegangen, dass Sie über Kenntnisse zu den grundlegenden Lambda-Operationen und der Lambda-Konsole verfügen. Sofern noch nicht geschehen, befolgen Sie die Anweisungen unter Erstellen einer Lambda-Funktion mit der Konsole, um Ihre erste Lambda-Funktion zu erstellen.

Um die folgenden Schritte durchzuführen, benötigen Sie die AWS CLI Version 2. Befehle und die erwartete Ausgabe werden in separaten Blöcken aufgeführt:

```
aws --version
```

Die Ausgabe sollte folgendermaßen aussehen:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

Bei langen Befehlen wird ein Escape-Zeichen (\) wird verwendet, um einen Befehl über mehrere Zeilen zu teilen.

Verwenden Sie auf Linux und macOS Ihren bevorzugten Shell- und Paket-Manager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine

Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux. Die CLI-Beispielbefehle in diesem Handbuch verwenden die Linux-Formatierung. Befehle, die Inline-JSON-Dokumente enthalten, müssen neu formatiert werden, wenn Sie die Windows-CLI verwenden.

Erstellen einer Ausführungsrolle

Erstellen Sie die Ausführungsrolle die Ihrer Lambda-Funktion die Berechtigung für den Zugriff auf AWS -Ressourcen erteilt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Rollen der AWS Identity and Access Management (IAM)-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften:
   - Vertrauenswürdiger Entitätstyp – AWS -Service
   - Anwendungsfall – Lambda
   - Berechtigungen – AWSLambdaBasicExecutionRole
   - Role name (Name der Rolle – **`response-streaming-role`**

Die AWSLambdaBasicExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion benötigt, um Protokolle in Amazon CloudWatch Logs zu schreiben. Nachdem Sie die Rolle erstellt haben, notieren Sie sich den Amazon-Ressourcennamen (ARN). Sie benötigen ihn im nächsten Schritt.

Erstellen Sie eine Antwort-Streaming-Funktion (AWS CLI)

Erstellen Sie eine Antwort-Streaming-Lambda-Funktion mit einem Funktions-URL-Endpunkt unter Verwendung der AWS Command Line Interface (AWS CLI).

Um eine Funktion zu erstellen, die Antworten streamen kann

1. Kopieren Sie das folgende Codebeispiel in eine Datei mit dem Namen `index.js`. Diese Funktion streamt drei Antworten im Abstand von 1 Sekunde.

```
exports.handler = awslambda.streamifyResponse(
 async (event, responseStream, _context) => {
```

```
  // Metadata is a JSON serializable JS object. Its shape is not defined here.
  const metadata = {
  statusCode: 200,
  headers: {
   "Content-Type": "application/json",
   "CustomHeader": "outerspace"
  }
  };

  // Assign to the responseStream parameter to prevent accidental reuse of the non-
wrapped stream.
  responseStream = awslambda.HttpResponseStream.from(responseStream, metadata);

  responseStream.write("Streaming with Helper \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 0 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 1 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 2 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.end();
  await responseStream.finished();
 }
  );
```

2. Erstellen Sie ein Bereitstellungspaket.

```
zip function.zip index.js
```

3. Erstellen Sie eine Lambda-Funktion mit dem Befehl `create-function`. Ersetzen Sie den Wert von `--role` durch den Rollen-ARN aus dem vorherigen Schritt. Mit diesem Befehl wird das Funktions-Timeout auf 10 Sekunden festgelegt, sodass die Funktion drei Antworten streamen kann.

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs24.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --timeout 10 \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

Um eine Funktions-URL zu erstellen

1. Fügen Sie Ihrer Funktion eine ressourcenbasierte Richtlinie hinzu, die die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` gewährt. Jede Anweisung muss in einem separaten Befehl hinzugefügt werden. Ersetzen Sie den Wert von `--principal` durch Ihre AWS-Konto ID.

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id UrlPolicyInvokeURL \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission \
    --function-name my-streaming-function \
    --action lambda:InvokeFunction \
    --statement-id UrlPolicyInvokeFunction \
    --principal 123456789012
```

2. Erstellen Sie einen URL-Endpunkt für die Funktion mit dem Befehl `create-function-url-config`.

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

> **Note**
>
> Wenn Sie eine Fehlermeldung über `--invoke-mode` erhalten, müssen Sie möglicherweise auf eine neuere Version der AWS CLI aktualisieren.

Testen des Funktions-URL-Endpunkts

Testen Sie Ihre Integration, indem Sie Ihre Funktion aufrufen. Sie können die URL Ihrer Funktion in einem Browser öffnen oder curl verwenden.

```
curl --request GET "https://abcdefghijklm7nop7qrs740abcd.lambda-url.us-east-1.on.aws/"
 --user "AKIAIOSFODNN7EXAMPLE" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

Unsere Funktions-URL nutzt den Authentifizierungstyp `IAM_AUTH`. Dies bedeutet, dass Sie Anfragen sowohl mit Ihrem AWS -Zugriffsschlüssel als auch mit Ihrem geheimen Schlüssel signieren müssen. Ersetzen Sie ihn im vorherigen Befehl `AKIAIOSFODNN7EXAMPLE` durch die AWS Zugriffsschlüssel-ID. Geben Sie Ihren AWS geheimen Schlüssel ein, wenn Sie dazu aufgefordert werden. Wenn Sie Ihren AWS geheimen Schlüssel nicht haben, können Sie stattdessen temporäre AWS Anmeldeinformationen verwenden.

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
Streaming with Helper
Hello 0
Hello 1
Hello 2
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Indem Sie AWS Ressourcen löschen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Verwenden des Lambda-Metadaten-Endpunkts

Mit dem Lambda-Metadaten-Endpunkt können Ihre Funktionen ermitteln, in welcher Availability Zone (AZ) sie ausgeführt werden. Sie können dies verwenden, um die Latenz zu optimieren, indem Sie zu denselben AZ-Ressourcen wie Amazon ElastiCache - und Amazon RDS-Endpunkten weiterleiten und Resilienzmuster implementieren AZ-aware .

Der Endpunkt gibt Metadaten in einem einfachen JSON-Format über eine Localhost-HTTP-API innerhalb der Ausführungsumgebung zurück und ist sowohl für Laufzeiten als auch für Erweiterungen zugänglich.

Sections

- Erste Schritte
- Grundlegendes zu Availability Zone IDs
- API-Referenz

Erste Schritte

Powertools for AWS Lambda bietet ein Hilfsprogramm für den Zugriff auf den Lambda-Metadaten-Endpunkt in Python TypeScript, Java und .NET. Das Hilfsprogramm speichert die Antwort nach dem ersten Aufruf im Cache und verarbeitet SnapStart die Cache-Invalidierung automatisch.

Verwenden Sie das Metadaten-Hilfsprogramm Powertools for AWS Lambda oder rufen Sie den Metadaten-Endpunkt direkt auf

Python

Installieren Sie das Powertools-Paket:

```
pip install "aws-lambda-powertools"
```

Verwenden Sie das Metadaten-Hilfsprogramm in Ihrem Handler:

Example Abrufen der AZ-ID mit Powertools (Python)

```
from aws_lambda_powertools.utilities.lambda_metadata import get_lambda_metadata

def handler(event, context):
    metadata = get_lambda_metadata()
```

```
    az_id = metadata.availability_zone_id  # e.g., "use1-az1"

    return {"az_id": az_id}
```

TypeScript

Installieren Sie das Powertools-Paket:

```
npm install @aws-lambda-powertools/commons
```

Verwenden Sie das Metadaten-Hilfsprogramm in Ihrem Handler:

Example Abrufen der AZ-ID mit Powertools () TypeScript

```
import { getMetadata } from '@aws-lambda-powertools/commons/utils/metadata';

const metadata = await getMetadata();

export const handler = async () => {
  const { AvailabilityZoneID: azId } = metadata;
  return azId;
};
```

Java

Fügen Sie die Powertools-Abhängigkeit zu Ihrem hinzu: `pom.xml`

```
<dependencies>
    <dependency>
        <groupId>software.amazon.lambda</groupId>
        <artifactId>powertools-lambda-metadata</artifactId>
        <version>2.10.0</version>
    </dependency>
</dependencies>
```

Verwenden Sie den Metadaten-Client in Ihrem Handler:

Example Abrufen der AZ-ID mit Powertools (Java)

```
import software.amazon.lambda.powertools.metadata.LambdaMetadata;
import software.amazon.lambda.powertools.metadata.LambdaMetadataClient;

public class App implements RequestHandler<Object, String> {
```

```
    @Override
    public String handleRequest(Object input, Context context) {
        LambdaMetadata metadata = LambdaMetadataClient.get();
        String azId = metadata.getAvailabilityZoneId(); // e.g., "use1-az1"

        return "{\"azId\": \"" + azId + "\"}";
    }
}
```

.NET

Installieren Sie das Powertools-Paket:

```
dotnet add package AWS.Lambda.Powertools.Metadata
```

Verwenden Sie die Metadatenklasse in Ihrem Handler:

Example Abrufen der AZ-ID mit Powertools (.NET)

```
using AWS.Lambda.Powertools.Metadata;

public class Function
{
    public string Handler(object input, ILambdaContext context)
    {
        var azId = LambdaMetadata.AvailabilityZoneId;
        return $"Running in AZ: {azId}";
    }
}
```

All Runtimes

Alle Lambda-Laufzeiten unterstützen den Metadaten-Endpunkt, einschließlich benutzerdefinierter Laufzeiten und Container-Images. Verwenden Sie das folgende Beispiel, um mithilfe der Umgebungsvariablen, die Lambda automatisch in der Ausführungsumgebung festlegt, direkt von Ihrer Funktion aus auf die Metadaten-API zuzugreifen.

Example Direkter Zugriff auf den Metadaten-Endpunkt

```
# Variables are automatically set by Lambda
METADATA_ENDPOINT="http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-
environment"
```

```
# Make the request
RESPONSE=$(curl -s -H "Authorization: Bearer ${AWS_LAMBDA_METADATA_TOKEN}"
 "$METADATA_ENDPOINT")

# Parse the AZ ID
AZ_ID=$(echo "$RESPONSE" | jq -r '.AvailabilityZoneID')

echo "Function is running in AZ ID: $AZ_ID"
```

Grundlegendes zu Availability Zone IDs

AZ-IDs (zum Beispiel`use1-az1`) beziehen sich immer auf denselben physischen Standort in allen AWS Konten, während AZ-Namen (z. B.`us-east-1a`) in bestimmten Regionen möglicherweise unterschiedlichen physischen Infrastrukturen in jedem AWS Konto zugeordnet werden. Weitere Informationen finden Sie unter AZ-IDs für kontoübergreifende Konsistenz.

Konvertierung der AZ-ID in einen AZ-Namen:

Verwenden Sie die Amazon EC2 DescribeAvailabilityZonesEC2-API, um eine AZ-ID in einen AZ-Namen zu konvertieren. Um diese API zu verwenden, fügen Sie die `ec2:DescribeAvailabilityZones` Berechtigung zur Ausführungsrolle Ihrer Funktion hinzu.

API-Referenz

Umgebungsvariablen

Lambda legt in jeder Ausführungsumgebung automatisch die folgenden Umgebungsvariablen fest:

- `AWS_LAMBDA_METADATA_API`— Die Metadaten-Serveradresse im Format `{ipv4_address}:{port}` (zum Beispiel`169.254.100.1:9001`).
- `AWS_LAMBDA_METADATA_TOKEN`— Ein eindeutiges Authentifizierungstoken für die aktuelle Ausführungsumgebung. Lambda generiert dieses Token automatisch bei der Initialisierung. Nehmen Sie es in alle Metadaten-API-Anfragen auf.

Endpoint

`GET http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-environment`

Anforderung

Erforderliche Header:

- `Authorization`— Der Tokenwert aus der `AWS_LAMBDA_METADATA_TOKEN` Umgebungsvariablen mit dem Bearer-Schema:. `Bearer <token>` Diese tokenbasierte Authentifizierung bietet umfassenden Schutz vor SSRF-Schwachstellen (Server-Side Request Forgery). Jede Ausführungsumgebung erhält bei der Initialisierung ein eindeutiges, zufällig generiertes Token.

Antwort

Status: `200 OK`

Content-Type: `application/json`

Cache-Control: `private, max-age=43200, immutable`

Die Antwort ist innerhalb einer Ausführungsumgebung unveränderlich. Clients sollten die Antwort zwischenspeichern und die `Cache-Control` TTL respektieren. Bei SnapStart Funktionen wird die TTL während der Initialisierung reduziert, sodass Clients die Metadaten nach der Wiederherstellung aktualisieren, wenn sich die Ausführungsumgebung möglicherweise in einer anderen AZ befindet. Wenn Sie Powertools verwenden, werden Caching und SnapStart Invalidierung automatisch durchgeführt.

Fließtext:

```
{
  "AvailabilityZoneID": "use1-az1"
}
```

Das `AvailabilityZoneID` Feld enthält den eindeutigen Bezeichner für die Availability Zone, in der die Ausführungsumgebung ausgeführt wird.

> **Note**
>
> In future Updates können der Antwort weitere Felder hinzugefügt werden. Kunden sollten unbekannte Felder ignorieren und nicht scheitern, wenn neue Felder erscheinen.

Fehlermeldungen

- 401 Nicht autorisiert — Der `Authorization` Header fehlt oder enthält ein ungültiges Token. Stellen Sie sicher, dass Sie die Prüfung bestanden haben`Bearer ${AWS_LAMBDA_METADATA_TOKEN}`.
- 405 Methode nicht zulässig — Die Anforderungsmethode ist nicht zulässig`GET`.
- 500 Interner Serverfehler — Server-side Verarbeitungsfehler.

Grundlegendes zu Aufrufmethoden für Lambda-Ereignisfunktionen

Nachdem Sie Ihre Lambda-Ereignisfunktion bereitgestellt haben, können Sie sie auf verschiedene Arten aufrufen:

- Die Lambda-Konsole – Verwenden Sie die Lambda-Konsole, um schnell ein Testereignis zu erstellen, mit dem Sie Ihre Funktion aufrufen.
- Das AWS SDK — Verwenden Sie das AWS SDK, um Ihre Funktion programmgesteuert aufzurufen.
- Die Invoke-API – Verwenden Sie die Lambda-Invoke-API, um Ihre Funktion direkt aufzurufen.
- Das AWS Command Line Interface (AWS CLI) — Verwenden Sie den `aws lambda invoke` AWS CLI Befehl, um Ihre Funktion direkt von der Befehlszeile aus aufzurufen.
- Ein Funktions-URL-HTTP(S)-Endpunkt – Verwenden Sie Funktions-URLs, um einen dedizierten HTTP(S)-Endpunkt zu erstellen, mit dem Sie Ihre Funktion aufrufen können.

All diese Methoden sind direkte Möglichkeiten, Ihre Event-Funktion aufzurufen. In Lambda besteht ein häufiger Anwendungsfall darin, Ihre Funktion auf der Grundlage eines Ereignisses aufzurufen, das an anderer Stelle in Ihrer Anwendung auftritt. Einige Dienste können bei jedem neuen Ereignis eine Lambda-Ereignisfunktion aufrufen.

Dies wird als Trigger bezeichnet. Für stream- und warteschlangenbasierte Dienste ruft Lambda die Funktion mit Batches von Datensätzen auf. Dies wird als Zuordnung von Ereignisquellen bezeichnet.

Wenn Sie eine Ereignisfunktion aufrufen, können Sie wählen, ob Sie sie synchron oder asynchron aufrufen möchten. Bei einem synchronen Aufruf warten Sie, bis die Funktion das Ereignis verarbeitet und eine Antwort zurückgegeben hat. Bei einem asynchronen Aufruf stellt Lambda das Ereignis für die Verarbeitung in eine Warteschlangen und gibt umgehend eine Antwort zurück. Der `InvocationType-Anforderungsparameter in der Invoke-API` bestimmt, wie Lambda Ihre Funktion

aufruft. Der Wert von `RequestResponse` steht für einen synchronen Aufruf und der Wert von `Event` steht für einen asynchronen Aufruf.

Verwenden Sie die öffentlichen Dual-Stack-Endpunkte von Lambda, um Ihre Event-Funktion über IPv6 aufzurufen. Dual-stack Endpunkte unterstützen sowohl IPv4 als auch IPv6. Lambda-Dual-Stack-Endpunkte verwenden die folgende Syntax:

```
protocol://lambda.us-east-1.api.aws
```

Sie können auch Lambda-Funktions-URLs verwenden, um Funktionen über IPv6 aufzurufen. Funktions-URL-Endpunkte haben das folgende Format:

```
https://url-id.lambda-url.us-east-1.on.aws
```

Wenn der Aufruf der Ereignisfunktion zu einem Fehler führt, sehen Sie sich bei synchronen Aufrufen die Fehlermeldung in der Antwort an und wiederholen Sie den Aufruf manuell. Bei asynchronen Aufrufen werden Wiederholungsversuche automatisch von Lambda verarbeitet und Aufrufdatensätze an ein Ziel gesendet.

Synchrones Aufrufen einer Lambda-Ereignisfunktion

Wenn Sie eine Lambda-Ereignisfunktion synchron aufrufen, führt Lambda die Funktion aus und wartet auf eine Antwort. Wenn der Funktionsaufruf beendet ist, gibt Lambda die Antwort aus dem Funktionscode mit zusätzlichen Daten, wie z. B. der Version der aufgerufenen Funktion, zurück. Verwenden Sie den Befehl, um eine Funktion synchron mit der aufzurufen. AWS CLI`invoke`

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

Das folgende Diagramm zeigt Clients, die synchron eine Lambda-Ereignisfunktion aufrufen. Lambda sendet die Ereignisse direkt an die Funktion und sendet die Antwort der Funktion zurück an den Aufrufer.



`payload` ist eine Zeichenfolge, die ein Ereignis im JSON-Format enthält. Der Name der Datei, in die die AWS CLI die Antwort von der Funktion schreibt, lautet `response.json`. Wenn die Ereignisfunktion ein Objekt oder einen Fehler zurückgibt, ist der Antworttext das Objekt oder der Fehler im JSON-Format. Wenn die Funktion fehlerfrei beendet wird, lautet der Antworttext `null`.

> **Note**
>
> Lambda wartet nicht, bis externe Erweiterungen abgeschlossen sind, bevor die Antwort gesendet wird. Externe Erweiterungen werden als unabhängige Prozesse in der Ausführungsumgebung ausgeführt und laufen auch nach Abschluss des Funktionsaufrufs weiter. Weitere Informationen finden Sie unter Erweitern Sie Lambda-Ereignisfunktionen mithilfe von Lambda-Erweiterungen.

Die Ausgabe des Befehls, die im Terminal angezeigt wird, enthält Informationen aus Headern in der Antwort von Lambda. Dies umfasst die Version, von der das Ereignis verarbeitet wurde (nützlich bei der Verwendung von Aliassen) und den von Lambda zurückgegebenen Statuscode. Wenn Lambda die Funktion ausführen konnte, lautet der Statuscode 200, auch wenn die Funktion einen Fehler zurückgegeben hat.

> **Note**
>
> Bei Ereignisfunktionen mit einem langen Timeout kann es sein, dass Ihr Client während eines synchronen Aufrufs unterbrochen wird, während er auf eine Antwort wartet. Konfigurieren Sie HTTP-Client, SDK, Firewall, Proxy oder Betriebssystem so, dass lange Verbindungen mit Timeout- oder Keepalive-Einstellungen möglich sind.

Wenn Lambda die Event-Funktion nicht ausführen kann, wird der Fehler in der Ausgabe angezeigt.

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload value response.json
```

Die Ausgabe sollte folgendermaßen aussehen:

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
```

```
 at [Source: (byte[])"value"; line: 1, column: 11]
```

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

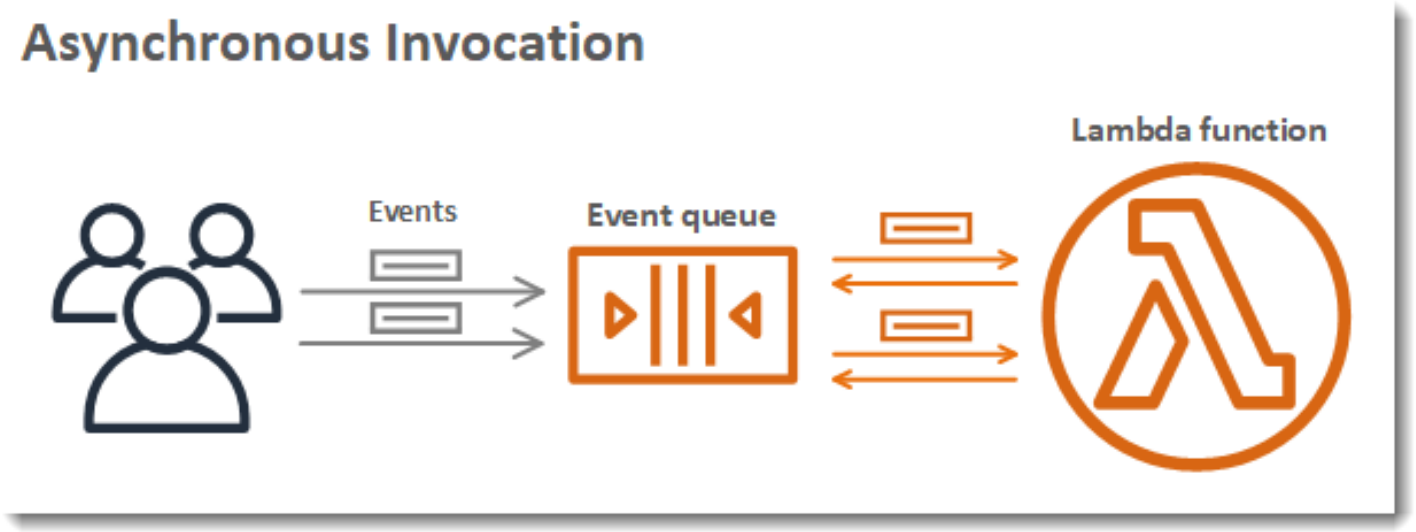
Weitere Informationen über die `Invoke`-API, einschließlich einer vollständigen Liste von Parametern, Kopfzeilen und Fehlern, finden Sie unter Aufrufen.

Wenn Sie eine Ereignisfunktion direkt aufrufen, können Sie die Antwort auf Fehler überprüfen und es erneut versuchen. Das AWS SDK AWS CLI und das SDK versuchen es auch bei Client-Timeouts, Drosselungen und Dienstfehlern automatisch erneut. Weitere Informationen finden Sie unter Grundlegendes zum Wiederholungsverhalten in Lambda-Ereignisfunktionen.

Asynchrones Aufrufen einer Lambda-Ereignisfunktion

Einige AWS-Services, wie Amazon Simple Storage Service (Amazon S3) und Amazon Simple Notification Service (Amazon SNS), rufen Event Functions asynchron auf, um Ereignisse zu verarbeiten. Sie können eine Lambda-Ereignisfunktion auch asynchron mit dem AWS Command Line Interface (AWS CLI) oder einem der SDKs aufrufen. AWS Wenn Sie eine Event-Funktion asynchron aufrufen, warten Sie nicht auf eine Antwort vom Funktionscode. Sie übergeben das Ereignis an Lambda und Lambda erledigt den Rest. Sie können konfigurieren, wie Lambda mit Fehlern umgeht, und Aufrufdatensätze an eine nachgelagerte Ressource wie Amazon Simple Queue Service (Amazon SQS) oder Amazon EventBridge (EventBridge) senden, um Komponenten Ihrer Anwendung miteinander zu verketten.

Das folgende Diagramm zeigt Clients, die eine Lambda-Funktion asynchron aufrufen. Lambda stellt die Ereignisse in eine Warteschlange, bevor sie an die Funktion gesendet werden.



Lambda platziert das Ereignis bei asynchronem Aufruf in eine Warteschlange und gibt eine Erfolgsantwort ohne zusätzliche Informationen zurück. Ein separater Prozess liest Ereignisse aus der Warteschlange und senden sie zu Ihrer Funktion.

Um eine Lambda-Ereignisfunktion asynchron mit dem AWS Command Line Interface (AWS CLI) oder einem der AWS SDKs aufzurufen, setzen Sie den Parameter auf. InvocationType`Event` Das folgende Beispiel zeigt einen AWS CLI Befehl zum Aufrufen einer Funktion.

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
```

```
  --payload '{ "key": "value" }' response.json
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 202
}
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabedatei (`response.json`) enthält keine Informationen, wird bei der Ausführung dieses Befehls aber dennoch erstellt. Wenn Lambda das Ereignis nicht zur Warteschlange hinzufügen kann, erscheint in der Befehlsausgabe eine Fehlermeldung.

Wie Lambda mit Fehlern und Wiederholungsversuchen mit asynchronem Aufruf umgeht

Lambda verwaltet Ihre asynchrone Ereigniswarteschlange der Funktion und unternimmt bei Fehlern Wiederholungsversuche. Wenn die Funktion einen Fehler zurückgibt, versucht Lambda standardmäßig zwei weitere Male, sie auszuführen. Dabei wird eine Minute lang zwischen den ersten zwei Versuchen und zwei Minuten lang zwischen dem zweiten und dem dritten Versuch gewartet. Zu den Eventfunktionsfehlern gehören Fehler, die vom Code der Funktion zurückgegeben werden, und Fehler, die von der Laufzeit der Funktion zurückgegeben werden, z. B. Timeouts.

Wenn für die Ereignisfunktion nicht genügend Parallelität zur Verfügung steht, um alle Ereignisse zu verarbeiten, werden zusätzliche Anfragen gedrosselt. Bei Drosselungsfehlern (429) und Systemfehlern (500-Serie) sendet Lambda das Ereignis zur Warteschlange zurück und versucht standardmäßig bis zu 6 Stunden lang, die Funktion erneut auszuführen. Das Wiederholungsintervall erhöht sich exponentiell von 1 Sekunde nach dem ersten Versuch auf maximal 5 Minuten. Lambda stellt viele Einträge in die Warteschlange, erhöht das Wiederholungsintervall und reduziert die Rate, mit der es Ereignisse aus der Warteschlange liest.

Selbst wenn Ihre Event-Funktion keinen Fehler zurückgibt, ist es möglich, dass sie dasselbe Ereignis mehrmals von Lambda empfängt, da die Warteschlange selbst irgendwann konsistent ist. Wenn die Funktion mit eingehenden Ereignissen nicht Schritt halten kann, kann es auch vorkommen, dass

Ereignisse aus der Warteschlange gelöscht werden, ohne zur Funktion gesendet zu werden. Stellen Sie sicher, dass Ihr Funktionscode doppelte Ereignisse ordnungsgemäß verarbeitet und dass Sie über genügend Gleichzeitigkeit zur Verarbeitung aller Aufrufe verfügen.

Wenn die Warteschlange sehr lang ist, können neue Ereignisse ablaufen, bevor Lambda sie an Ihre Event-Funktion senden kann. Wenn ein Ereignis abläuft oder bei ihm alle Verarbeitungsversuche fehlschlagen, wird es von Lambda verworfen. Sie können die Fehlerbehandlung für eine Event-Funktion konfigurieren, um die Anzahl der Wiederholungen zu reduzieren, die Lambda durchführt, oder um unverarbeitete Ereignisse schneller zu verwerfen. Konfigurieren Sie eine Warteschlange für unzustellbare Nachrichten, um verworfene Ereignisse zu erfassen. Zur Erfassung von Aufzeichnungen fehlgeschlagener Aufrufe (z. B. Zeitüberschreitungen oder Laufzeitfehler) zu erfassen, erstellen Sie Fehlerziel.

Konfigurieren von Fehlerbehandlungseinstellungen für asynchrone Lambda-Aufrufe

Verwenden Sie die folgenden Einstellungen, um zu konfigurieren, wie Lambda Fehler und Wiederholungen bei asynchronen Funktionsaufrufen behandelt:

- MaximumEventAgeInSeconds: Die maximale Zeitspanne in Sekunden, für die Lambda ein Ereignis in der asynchronen Ereigniswarteschlange aufbewahrt, bevor es verworfen wird.
- MaximumRetryAttempts: Die maximale Häufigkeit, mit der Lambda Ereignisse wiederholt, wenn die Funktion einen Fehler zurückgibt.

Verwenden Sie die Lambda-Konsole oder AWS CLI um Einstellungen zur Fehlerbehandlung für eine Event-Funktion, eine Version oder einen Alias zu konfigurieren.

Console

So konfigurieren Sie die Fehlerbehandlung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie „Konfiguration“ und dann „Asynchroner Aufruf“ aus.
4. Wählen Sie unter Asynchronous invocation (Asynchroner Aufruf) die Option Edit (Bearbeiten).
5. Konfigurieren Sie die folgenden Einstellungen.

    - Maximales Alter des Ereignisses – Die maximale Zeitspanne, die Lambda ein Ereignis in der Warteschlange für asynchrone Ereignisse behält (bis zu 6 Stunden).

- Wiederholversuche – Die Anzahl der Wiederholungen, die Lambda es erneut versucht, wenn die Funktion einen Fehler zurückgibt (zwischen 0 und 2).

6. Wählen Sie Speichern.

AWS CLI

Verwenden Sie den Befehl put-function-event-invoke-config AWS CLI, um den asynchronen Aufruf mit dem zu konfigurieren. Im folgenden Beispiel wird eine Ereignisfunktion mit einem maximalen Ereignisalter von 1 Stunde und ohne Wiederholungsversuche konfiguriert.

```
aws lambda put-function-event-invoke-config \
  --function-name error \
  --maximum-event-age-in-seconds 3600 \
  --maximum-retry-attempts 0
```

Der Befehl `put-function-event-invoke-config` überschreibt alle vorhandenen Konfigurationen für die Funktion, die Version oder den Alias. Um eine Option zu konfigurieren, ohne andere zurückzusetzen, verwenden Sie update-function-event-invoke-config. Im folgenden Beispiel wird Lambda so konfiguriert, dass ein Datensatz an eine SQS-Standardwarteschlange namens `destination` gesendet wird, wenn ein Ereignis nicht verarbeitet werden kann.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
```

```
}
```

Wenn ein Aufrufereignis das maximale Alter überschreitet oder alle Wiederholversuche fehlschlagen, wird es von Lambda verworfen. Um eine Kopie der verworfenen Ereignisse beizubehalten, konfigurieren Sie ein Ziel für fehlgeschlagene Ereignisse.

Erfassung von Datensätzen asynchroner Lambda-Aufrufe

Lambda kann Aufzeichnungen von asynchronen Aufrufen an einen der folgenden Server senden. AWS-Services

- Amazon SQS – Eine standardmäßige SQS-Warteschlange
- Amazon SNS – Ein SNS-Standardthema
- Amazon S3 – Ein Amazon S3-Bucket (nur bei einem Fehler)
- AWS Lambda – Eine Lambda-Funktion
- Amazon EventBridge — Ein EventBridge Eventbus

Der Aufrufdatensatz enthält Details zur Anforderung und Antwort im JSON-Format. Sie können separate Ziele für Ereignisse konfigurieren, die erfolgreich verarbeitet werden, und für Ereignisse, bei denen alle Verarbeitungsversuche fehlschlagen. Alternativ können Sie eine Amazon-SQS-Standardwarteschlange oder ein Amazon-SNS-Standardthema als Warteschlange für unzustellbare Nachrichten für verworfene Ereignisse konfigurieren. Für Warteschlangen für unzustellbare Nachrichten sendet Lambda nur den Inhalt des Ereignisses ohne Details zur Antwort.

Wenn Lambda einen Datensatz nicht an ein von Ihnen konfiguriertes Ziel senden kann, sendet es eine `DestinationDeliveryFailures` Metrik an Amazon CloudWatch. Dies kann passieren, wenn Ihre Konfiguration einen nicht unterstützten Zieltyp enthält, z. B. eine Amazon-SQS-FIFO-Warteschlange oder ein Amazon-SNS-FIFO-Thema. Zustellungsfehler können auch aufgrund von Berechtigungsfehlern und Größenbeschränkungen auftreten. Weitere Informationen zu Lambda-Aufrufmetriken finden Sie unter the section called “Aufrufmetriken”.

> **Note**
>
> Um zu verhindern, dass eine Event-Funktion ausgelöst wird, können Sie die reservierte Parallelität der Funktion auf Null setzen. Wenn Sie die reservierte Parallelität für eine asynchron aufgerufene Funktion auf Null setzen, sendet Lambda sofort und ohne Wiederholungsversuche alle Ereignisse ohne erneute Versuche an die konfigurierte

Warteschlange für unzustellbare Nachrichten oder an das Ereignisziel bei Fehlern. Um Ereignisse zu verarbeiten, die gesendet wurden, während die reservierte Parallelität auf Null gesetzt war, müssen Sie die Ereignisse aus der Warteschlange für unzustellbare Nachrichten oder dem Ereignisziel bei Fehlern verarbeiten.

Hinzufügen eines Ziels

Um Aufzeichnungen über asynchrone Aufrufe beizubehalten, fügen Sie Ihrer Event-Funktion ein Ziel hinzu. Sie können wählen, ob erfolgreiche Aufrufe oder fehlgeschlagene Aufrufe an ein Ziel gesendet werden sollen. Jede Funktion kann mehrere Ziele haben, sodass Sie separate Ziele für erfolgreiche und fehlgeschlagene Ereignisse konfigurieren können. Jeder Datensatz, der an das Ziel gesendet wird, ist ein JSON-Dokument mit Details zum Aufruf. Wie bei den Einstellungen für die Fehlerbehandlung können Sie Ziele für eine Ereignisfunktion, eine Funktionsversion oder einen Alias konfigurieren.

Tip

Sie können auch Aufzeichnungen über fehlgeschlagene Aufrufe für die folgenden Typen von Ereignisquellenzuordnungen speichern: Amazon Kinesis, Amazon DynamoDB und Apache Kafka (Amazon MSK und selbstverwalteter Apache Kafka).

In der folgenden Tabelle werden die unterstützten Ziele für Datensätze der asynchronen Aufrufen aufgelistet. Damit Lambda erfolgreich Datensätze an das von Ihnen gewählte Ziel senden kann, stellen Sie sicher, dass die Ausführungsrolle Ihrer Event-Funktion auch die entsprechenden Berechtigungen enthält. In der Tabelle wird auch beschrieben, wie jeder Zieltyp den JSON-Aufrufdatensatz empfängt.

Zieltyp	Erforderliche Berechtigung	Destination-specific JSON-Format
Amazon-SQS-Warteschlange	sqs: SendMessage	Lambda übergibt den Aufrufdatensatz als `Message` an das Ziel.

Zieltyp	Erforderliche Berechtigung	Destination-specific JSON-Format
Amazon-SNS-Thema	sns:Publish	Lambda übergibt den Aufrufdatensatz als `Message` an das Ziel.
Ein Amazon S3-Bucket (nur bei einem Fehler)	s3: PutObject s3: ListBucket	• Lambda speichert den Aufrufdatensatz als JSON-Objekt im Ziel-Bucket. • Der Name des S3-Objekts wird nach folgender Konvention benannt: `aws/lambda/async/<function-name>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>`
Lambda-Ereignisfunktion	Lambda: InvokeFunction	Lambda übergibt den Aufrufdatensatz als Nutzlast in der Funktion.

Zieltyp	Erforderliche Berechtigung	Destination-specific JSON-Format
EventBridge	Ereignisse: PutEvents	• Lambda übergibt den Aufrufdatensatz wie `detail` im PutEvents Aufruf. • Der Wert für das `source`-Ereignisfeld ist `lambda`. • Der Wert für das `detail-type` -Ereignisfeld ist entweder „Lambda Function Invocation Result – Success“ (Ergebnis des Lambda-Funktionsaufrufs: Erfolgreich) oder „Lambda Function Invocation Result – Failure“ (Ergebnis des Lambda-Funktionsaufrufs: Fehler). • Das `resource`-Ereignis feld enthält die Amazon Resource Names (ARNs) für Funktion und Ziel. • Weitere Veranstaltungsbereiche finden Sie unter EventBridge Amazon-Veranstaltungen.

> **Note**
>
> Wenn Sie für Amazon S3 S3-Ziele die Verschlüsselung für den Bucket mithilfe eines KMS-Schlüssels aktiviert haben, benötigt Ihre Event Function auch die kms: GenerateDataKey -Berechtigung.

> **Important**
>
> Wenn Sie Amazon SNS als Ziel verwenden, beachten Sie, dass Amazon SNS eine maximale Nachrichtengröße von 256 KB hat. Wenn sich Ihre Nutzlast für asynchrone Aufrufe 1 MB nähert, kann der Aufrufdatensatz (der die ursprüngliche Nutzlast plus zusätzliche Metadaten enthält) das Amazon SNS SNS-Limit überschreiten und zu Zustellungsfehlern führen. Erwägen Sie die Verwendung von Amazon-SQS- oder Amazon-S3-Zielen für größere Nutzdaten.

In den folgenden Schritten wird beschrieben, wie Sie mit der Lambda-Konsole und der AWS CLI ein Ziel für eine Funktion konfigurieren.

Console

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie unter Source (Quelle) die Option Asynchronous invocation (Asynchroner Aufruf).
5. Wählen Sie unter Condition (Bedingung) eine der folgenden Optionen aus:
   - Bei Fehler – Es wird ein Datensatz gesendet, wenn das Ereignis alle Verarbeitungsversuche ausschöpft oder das maximale Alter überschreitet.
   - Bei Erfolg – Es wird ein Datensatz gesendet, wenn die Funktion einen asynchronen Aufruf erfolgreich verarbeitet.
6. Wählen Sie unter Destination type (Zieltyp) den Ressourcentyp aus, der den Aufrufdatensatz empfängt.
7. Wählen Sie unter Destination (Ziel) eine Ressource aus.
8. Wählen Sie Speichern.

AWS CLI

Um ein Ziel mit dem zu konfigurieren AWS CLI, führen Sie den Befehl update-function-event-invoke-config aus. Im folgenden Beispiel wird Lambda so konfiguriert, dass ein Datensatz an eine

SQS-Standardwarteschlange namens `destination` gesendet wird, wenn ein Ereignis nicht verarbeitet werden kann.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Bewährte Methoden für die Sicherheit in Amazon-S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze aus Ihrer Event-Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die Berechtigungen auf Buckets in Ihrem Konto beschränkt. `s3:PutObject`

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
```

```
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Lambda-Ereignisfunktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:
    a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.
    b. Wählen Sie im Richtlinien-Editor JSON aus.
    c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.
    d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.
    e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle (CLI) einer Event-Funktion eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Beispiel für Aufrufdatensatz

Wenn ein Aufruf mit der Bedingung übereinstimmt, sendet Lambda ein JSON-Dokument mit Details zum Aufruf an das Ziel. Das folgende Beispiel zeigt einen Aufrufdatensatz für ein Ereignis, bei dem drei Verarbeitungsversuche aufgrund eines Fehlers in der Ereignisfunktion fehlgeschlagen sind.

Example

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
```

```
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

Der Aufrufdatensatz enthält Details zum Ereignis, zur Antwort und zu dem Grund, warum der Datensatz gesendet wurde.

Zurückverfolgen von Anforderungen zu Zielen

Sie können AWS X-Ray damit eine verbundene Ansicht jeder Anfrage anzeigen, während sie sich in der Warteschlange befindet, von einer Lambda-Ereignisfunktion verarbeitet und an den Zieldienst übergeben wird. Wenn Sie die X-Ray Ablaufverfolgung für eine Funktion oder einen Dienst aktivieren, der eine Funktion aufruft, fügt Lambda der Anfrage einen X-Ray Header hinzu und übergibt den Header an den Zieldienst. Traces von Upstream-Diensten werden automatisch mit Traces von Downstream-Lambda-Ereignisfunktionen und Zieldiensten verknüpft, wodurch eine umfassende Ansicht der gesamten Anwendung entsteht. Weitere Informationen zum Tracing finden Sie unter Visualisieren Sie Aufrufe von Lambda-Ereignisfunktionen mit AWS X-Ray.

Hinzufügen einer Warteschlange für unzustellbare Nachrichten

Als Alternative zu einem Ziel für den Fall eines Fehlers können Sie Ihre Event-Funktion mit einer Warteschlange für unveröffentlichte Nachrichten konfigurieren, um verworfene Ereignisse für die weitere Verarbeitung zu speichern. Eine Warteschlange für unzustellbare Nachrichten fungiert genauso wie ein Ziel bei Ausfall, da sie verwendet wird, wenn alle Verarbeitungsversuche eines Ereignisses fehlschlagen oder ein Ereignis abläuft, ohne verarbeitet zu werden. Sie können jedoch nur auf Funktionsebene eine Warteschlange für unzustellbare Nachrichten hinzufügen oder entfernen. Versionen von Event-Funktionen verwenden dieselben Einstellungen für Warteschlangen mit unveröffentlichten Nachrichten ($LATEST). On-failure Ziele unterstützen auch zusätzliche Ziele und enthalten Details zur Antwort der Funktion im Aufrufdatensatz.

Um Ereignisse in einer Warteschlange mit toten Buchstaben erneut zu verarbeiten, können Sie sie als Ereignisquelle für Ihre Lambda-Ereignisfunktion festlegen. Alternativ können Sie die Ereignisse manuell abrufen.

Sie können eine Amazon-SQS-Standardwarteschlange oder ein Amazon-SNS-Standardthema für Ihre Warteschlange für unzustellbare Nachrichten auswählen. FIFO-Warteschlangen und Amazon-SNS-FIFO-Themen werden nicht unterstützt.

- Amazon-SQS-Warteschlange – Eine Warteschlange nimmt fehlgeschlagene Ereignisse auf, bis sie abgerufen werden. Wählen Sie eine Amazon SQS SQS-Standardwarteschlange, wenn Sie erwarten, dass eine einzelne Entität, z. B. eine Lambda-Ereignisfunktion oder ein CloudWatch Alarm, das fehlgeschlagene Ereignis verarbeitet. Weitere Informationen finden Sie unter Verwenden von Lambda mit Amazon SQS.
- Amazon-SNS-Thema – Ein Thema leitet fehlgeschlagene Ereignisse an eine oder mehrere Ziele. Wählen Sie ein Amazon-SNS-Standardthema aus, wenn Sie erwarten, dass mehrere Entitäten auf ein fehlgeschlagenes Ereignis reagieren. Sie können beispielsweise ein Thema so konfigurieren, dass Ereignisse an eine E-Mail-Adresse, eine Lambda-Ereignisfunktion oder and/or einen HTTP-Endpunkt gesendet werden. Weitere Informationen finden Sie unter Aufrufen von Lambda-Ereignisfunktionen mit Amazon SNS SNS-Benachrichtigungen.

Um Ereignisse an eine Warteschlange oder ein Thema zu senden, benötigt Ihre Event-Funktion zusätzliche Berechtigungen. Fügen Sie eine Richtlinie mit den erforderlichen Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzu. Wenn die Zielwarteschlange oder das Zielthema mit einem vom Kunden verwalteten AWS KMS Schlüssel verschlüsselt ist, stellen Sie sicher, dass sowohl die Ausführungsrolle Ihrer Funktion als auch die ressourcenbasierte Richtlinie des Schlüssels die entsprechenden Berechtigungen enthalten.

Nachdem Sie das Ziel erstellt und die Ausführungsrolle Ihrer Event-Funktion aktualisiert haben, fügen Sie Ihrer Funktion die Warteschlange mit unerlaubten Briefen hinzu. Sie können mehrere Funktionen zum Senden von Ereignissen an dasselbe Ziel konfigurieren.

Console

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie „Konfiguration“ und dann „Asynchroner Aufruf“ aus.
4. Wählen Sie unter Asynchronous invocation (Asynchroner Aufruf) die Option Edit (Bearbeiten).

5. Stellen Sie den Dead-letter Warteschlangenservice auf Amazon SQS oder Amazon SNS ein.
6. Wählen Sie die Ziel-Warteschlange oder das Ziel-Thema aus.
7. Wählen Sie Speichern.

AWS CLI

Verwenden Sie den Befehl update-function-configuration, um eine Warteschlange mit unerlaubten AWS CLI Briefen zu konfigurieren.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --dead-letter-config TargetArn=arn:aws:sns:us-east-1:123456789012:my-topic
```

Lambda sendet das Ereignis mit zusätzlichen Informationen in den Attributen an die Warteschlange für unzustellbare Nachrichten. Sie können anhand dieser Informationen den von der Funktion zurückgegebenen Fehler erkennen oder das Ereignis mit Protokollen oder einer AWS X-Ray - Ablaufverfolgung korrelieren.

Dead-letter Nachrichtenattribute in die Warteschlange

- RequestID (Zeichenfolge) – Die ID der Aufrufanforderung. Anforderungs-IDs erscheinen in Funktionsprotokollen. Sie können das X-Ray SDK auch verwenden, um die Anforderungs-ID für ein Attribut im Trace aufzuzeichnen. Sie können dann anhand der Anforderungs-ID in der X-Ray Konsole nach Traces suchen.
- ErrorCode(Nummer) — Der HTTP-Statuscode.
- ErrorMessage(String) — Die ersten 1 KB der Fehlermeldung.

Wenn Lambda keine Nachricht an die Warteschlange für unzustellbare Briefe senden kann, löscht es das Ereignis und gibt die Metrik aus. DeadLetterErrors Dies kann bei mangelnden Berechtigungen passieren oder wenn die Gesamtgröße der Nachricht den Grenzwert der Zielwarteschlange oder des Ziel-Themas überschreitet. Nehmen wir zum Beispiel an, dass eine Amazon-SNS-Benachrichtigung mit einem Text von fast 1 MB eine Funktion auslöst, die zu einem Fehler führt. In diesem Fall können die Ereignisdaten, die Amazon SNS hinzufügt, in Kombination mit den Attributen, die Lambda hinzufügt, dazu führen, dass die Nachricht die maximale Größe überschreitet, die in der Warteschlange für unzustellbare Nachrichten zulässig ist.

Wenn Sie Amazon SQS als Ereignisquelle verwenden, konfigurieren Sie eine Warteschlange für unzustellbare Briefe in der Amazon SQS SQS-Warteschlange selbst und nicht in der Lambda Event Function. Weitere Informationen finden Sie unter Verwenden von Lambda mit Amazon SQS.

Wie Lambda Datensätze aus Stream- und warteschlangenbasierten Ereignisquellen verarbeitet

Ein Zuordnung von Ereignisquellen ist eine Lambda-Ressource, die Elemente aus Stream- und Warteschlangen-basierten Diensten liest und eine Funktion mit Stapeln von Datensätzen aufruft. Innerhalb einer Zuordnung von Ereignisquellen fragen Ressourcen, sogenannte Event-Poller, aktiv nach neuen Nachrichten ab und rufen Funktionen auf. Standardmäßig skaliert Lambda Ereignisabfragen automatisch, aber für bestimmte Ereignisquellentypen können Sie den Bereitstellungsmodus verwenden, um die Mindest- und Höchstanzahl von Event-Pollern zu steuern, die für Ihre Zuordnung von Ereignisquellen vorgesehen sind.

Die folgenden Dienste verwenden Ereignisquellenzuordnungen, um Lambda-Ereignisfunktionen aufzurufen:

- Amazon DocumentDB (mit MongoDB-Kompatibilität) (Amazon DocumentDB)
- Amazon-DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon-Simple-Queue-Service (Amazon SQS)

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Wie unterscheiden sich Zuordnungen von Ereignisquellen von direkten Auslösern

Einige AWS-Services können Lambda-Ereignisfunktionen mithilfe von Triggern direkt aufrufen. Diese Dienste leiten Ereignisse an Lambda weiter und die Funktion wird sofort aufgerufen, wenn das angegebene Ereignis eintritt. Trigger eignen sich für diskrete Ereignisse und die Verarbeitung in

Echtzeit. Wenn Sie mit der Lambda-Konsole einen Trigger erstellen, interagiert die Konsole mit dem entsprechenden AWS Dienst, um die Ereignisbenachrichtigung für diesen Dienst zu konfigurieren. Der Auslöser wird tatsächlich von dem Dienst gespeichert und verwaltet, der die Ereignisse generiert, nicht von Lambda. Hier sind einige Beispiele für Dienste, die Trigger verwenden, um Lambda-Ereignisfunktionen aufzurufen:

- Amazon Simple Storage Service (Amazon S3): Ruft eine Funktion auf, wenn ein Objekt in einem Bucket erstellt, gelöscht oder geändert wird. Weitere Informationen finden Sie unter Tutorial: Verwenden eines Amazon S3 S3-Triggers zum Aufrufen einer Lambda-Ereignisfunktion.
- Amazon Simple Notification Service (Amazon SNS): Ruft eine Funktion auf, wenn eine Nachricht in einem SNS-Thema veröffentlicht wird. Weitere Informationen finden Sie unter Tutorial: Verwenden AWS Lambda mit Amazon Simple Notification Service.
- Amazon API Gateway: Ruft eine Funktion auf, wenn eine API-Anfrage an einen bestimmten Endpunkt gestellt wird. Weitere Informationen finden Sie unter Aufrufen einer Lambda-Ereignisfunktion mithilfe eines Amazon API Gateway Gateway-Endpunkts.

Zuordnungen von Ereignisquellen sind Lambda-Ressourcen, die innerhalb des Lambda-Service erstellt und verwaltet werden. Zuordnungen von Ereignisquellen sind für die Verarbeitung umfangreicher Streaming-Daten oder Nachrichten aus Warteschlangen konzipiert. Die stapelweise Verarbeitung von Datensätzen aus einem Stream oder einer Warteschlange ist effizienter als die Verarbeitung einzelner Datensätze.

Batching-Verhalten

Standardmäßig fasst eine Ereignisquellenzuordnung Datensätze zu einer einzigen Nutzlast zusammen, die Lambda an Ihre Event-Funktion sendet. Um das Batching-Verhalten zu optimieren, können Sie ein Batching-Fenster (MaximumBatchingWindowInSeconds) und eine Batch-Größe (BatchSize) konfigurieren. Ein Batch-Fenster ist die maximale Zeitspanne zur Erfassung von Datensätzen in einer einzigen Nutzlast. Eine Batch-Größe ist die maximale Anzahl von Datensätzen in einem einzigen Batch. Lambda ruft Ihre Funktion auf, wenn eines der folgenden drei Kriterien erfüllt ist:

- Das Batching-Fenster erreicht seinen Maximalwert. Das standardmäßige Batch-Fensterverhalten unterscheidet sich je nach Ereignisquelle.
  - Für Kinesis-, DynamoDB- und Amazon SQS SQS-Ereignisquellen: Das Standard-Batch-Fenster beträgt 0 Sekunden. Das bedeutet, dass Lambda Ihre Funktion aufruft, sobald Datensätze verfügbar sind. Um ein Batching-Fenster festzulegen, konfigurieren Sie

`MaximumBatchingWindowInSeconds`. Sie können diesen Parameter auf einen beliebigen Wert zwischen 0 und 300 Sekunden in Schritten von 1 Sekunde einstellen. Wenn Sie ein Stapelverarbeitungsfenster konfigurieren, beginnt das nächste Fenster, sobald der vorherige Funktionsaufruf abgeschlossen ist.

- Für Amazon-MSK-, selbstverwaltete Apache-Kafka-, Amazon-MQ- und Amazon-DocumentDB-Ereignisquellen: Das standardmäßige Batching-Fenster beträgt 500 ms. Sie können `MaximumBatchingWindowInSeconds` auf einen beliebigen Wert von 0 Sekunden bis 300 Sekunden in Sekundenschritten einstellen. Wenn Sie im Bereitstellungsmodus für Kafka-Zuordnungen von Ereignisquellen ein Batching-Fenster konfigurieren, beginnt das nächste Fenster, sobald der vorherige Batch abgeschlossen ist. Wenn Sie ein Batching-Fenster konfigurieren, beginnt das nächste Fenster, sobald der vorherige Funktionsaufruf abgeschlossen ist.

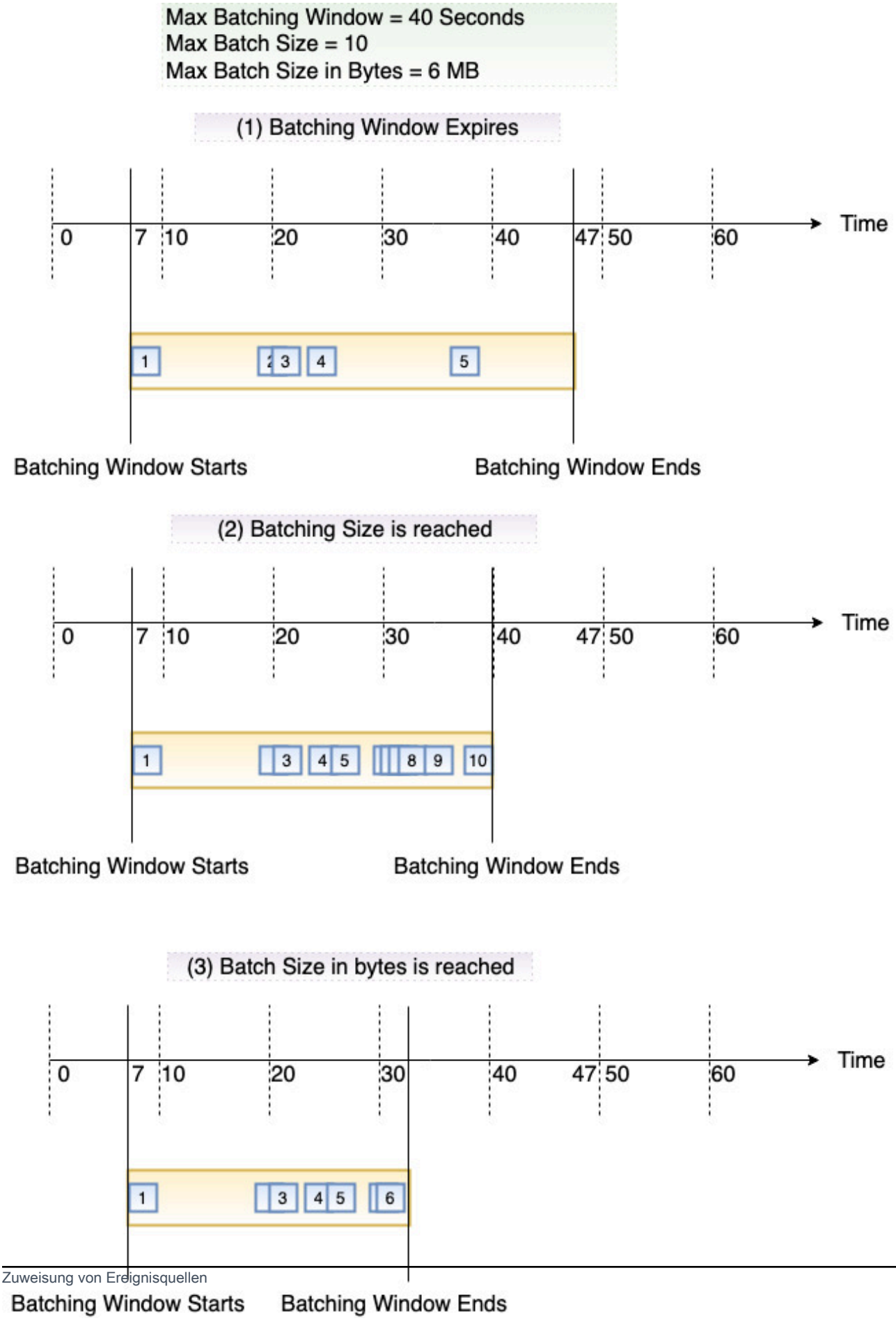
  Um die Latenz bei der Verwendung von Kafka-Zuordnungen von Ereignisquellen im Bereitstellungsmodus zu minimieren, setzen Sie `MaximumBatchingWindowInSeconds` auf 0. Diese Einstellung stellt sicher, dass Lambda unmittelbar nach Abschluss des aktuellen Funktionsaufrufs mit der Verarbeitung des nächsten Batches beginnt. Weitere Informationen zur Verarbeitung mit niedriger Latenz finden Sie unter Niedrige Latenz für Apache-Kafka.

- Für die Ereignisquellen Amazon MQ und Amazon DocumentDB: Das standardmäßige Batching-Fenster beträgt 500 ms. Sie können `MaximumBatchingWindowInSeconds` auf einen beliebigen Wert von 0 Sekunden bis 300 Sekunden in Sekundenschritten einstellen. Ein Batch-Fenster beginnt, sobald der erste Datensatz eintrifft.

  > **Note**
  >
  > Da Sie `MaximumBatchingWindowInSeconds` nur in Sekundenschritten ändern können, können Sie nicht zu dem Standard-Batch-Fenster von 500 ms zurückkehren, nachdem Sie es geändert haben. Um das Standard-Batch-Fenster wiederherzustellen, müssen Sie eine neue Ereignisquellenzuordnung erstellen.

- Die Batch-Größe wird erreicht. Die minimale Batch-Größe beträgt 1. Die Standard- und die maximale Batch-Größe hängen von der Ereignisquelle ab. Weitere Informationen zu diesen Werten finden Sie unter BatchSize-Spezifikation für die `CreateEventSourceMapping`-API-Operation.
- Die Nutzlastgröße erreicht 6 MB. Sie können dieses Limit nicht ändern.

Das folgende Diagramm verdeutlicht diese Bedingungen. Angenommen, ein Batch-Fenster beginnt bei `t = 7` Sekunden. Im ersten Szenario erreicht das Batch-Fenster sein Maximum von 40 Sekunden bei `t = 47` Sekunden nach dem Erfassen von 5 Datensätzen. Im zweiten Szenario erreicht die Batch-Größe 10, bevor das Batch-Fenster abläuft, sodass das Batch-Fenster früh endet. Im dritten Szenario wird die maximale Nutzlastgröße erreicht, bevor das Batch-Fenster abläuft, sodass das Batch-Fenster frühzeitig endet.


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

Wir empfehlen, dass Sie mit unterschiedlichen Stapel- und Datensatzgrößen testen, sodass die Abfragefrequenz der einzelnen Ereignisquellen darauf abgestimmt ist, wie schnell Ihre Ereignisfunktion ihre Aufgabe erledigen kann. Der CreateEventSourceMapping`BatchSize`Parameter steuert die maximale Anzahl von Datensätzen, die bei jedem Aufruf an Ihre Funktion gesendet werden können. Eine höhere Stapelgröße kann den mit dem Aufruf-Overhead über eine größere Datensatzgruppe hinweg oft effizienter verarbeiten und Ihren Durchsatz erhöhen.

Lambda wartet mit dem Senden des nächsten zu verarbeitenden Stapels nicht, bis ggf. konfigurierte Erweiterungen abgeschlossen sind. Anders ausgedrückt: Ihre Erweiterungen werden möglicherweise weiter ausgeführt, während Lambda den nächsten Stapel von Datensätzen verarbeitet. Dies kann zu Drosselungsproblemen führen, wenn Sie gegen eine Einstellung oder gegen einen Grenzwert im Zusammenhang mit der Parallelität Ihres Kontos verstoßen. Um zu erkennen, ob möglicherweise ein Problem vorliegt, müssen Sie Ihre Funktionen überwachen sowie überprüfen, ob für Ihre Zuordnung von Ereignisquellen unerwartet hohe Parallelitätsmetriken vorliegen. Aufgrund der kurzen Zeit zwischen den Aufrufen kann Lambda kurzzeitig eine höhere Gleichzeitigkeitsnutzung als die Anzahl der Shards melden. Dies kann sogar für Lambda Event Functions ohne Erweiterungen gelten.

Wenn Ihre Funktion einen Fehler zurückgibt, verarbeitet die Ereignisquellenzuordnung standardmäßig das gesamte Batch erneut, bis die Funktion erfolgreich ist oder die Elemente im Batch ablaufen. Um eine ordnungsgemäße Verarbeitung zu gewährleisten, unterbricht die Ereignisquellenzuordnung die Verarbeitung für den betroffenen Shard, bis der Fehler behoben ist. Für Stream-Quellen (DynamoDB und Kinesis) können Sie die maximale Anzahl von Wiederholungsversuchen von Lambda konfigurieren, wenn Ihre Funktion einen Fehler zurückgibt. Service-Fehler oder Drosselungen, bei denen der Stapel Ihre Funktion nicht erreicht, zählen nicht zu den Wiederholungsversuchen. Sie können die Zuordnung von Ereignisquellen auch so konfigurieren, dass ein Aufrufdatensatz an ein Ziel gesendet wird, wenn ein Ereignisbatch verworfen wird.

Modus bereitgestellter Kapazität

Lambda-Ereignisquellenzuordnungen verwenden Ereignisabfragen, um Ihre Ereignisquelle nach neuen Nachrichten abzufragen. Standardmäßig verwaltet Lambda die automatische Skalierung dieser Poller auf der Grundlage des Nachrichtenvolumens. Wenn der Nachrichtenverkehr zunimmt, erhöht Lambda automatisch die Anzahl der Event-Poller, um die Last zu bewältigen und reduziert sie, wenn der Verkehr abnimmt.

Im Bereitstellungsmodus können Sie den Durchsatz Ihrer Ereignisquellenzuordnung feinabstimmen, indem Sie Mindest- und Höchstgrenzen für dedizierte Abfrageressourcen definieren, die bereit sind, die erwarteten Datenverkehrsmuster zu verarbeiten. Diese Ressourcen werden dreimal schneller automatisch skaliert, um plötzliche Spitzen im Ereignisverkehr zu bewältigen, und bieten eine 16-

mal höhere Kapazität zur Verarbeitung von Millionen von Ereignissen. Auf diese Weise können Sie reaktionsschnelle ereignisgesteuerte Workloads mit strengen Leistungsanforderungen erstellen.

In Lambda ist ein Event Poller eine Recheneinheit mit Durchsatzmöglichkeiten, die je nach Art der Ereignisquelle variieren. Bei Amazon MSK und selbstverwaltetem Apache Kafka kann jeder Event Poller bis zu 5% MB/sec des Durchsatzes oder bis zu 5 gleichzeitige Aufrufe verarbeiten. Wenn Ihre Ereignisquelle beispielsweise eine durchschnittliche Nutzlast von 1 MB erzeugt und die durchschnittliche Dauer Ihrer Funktion 1 Sekunde beträgt, kann ein einziger Kafka-Event-Poller 5 MB/sec Durchsatz- und 5 gleichzeitige Lambda-Aufrufe unterstützen (vorausgesetzt, es gibt keine Payload-Transformation). Für Amazon SQS kann jeder Event Poller bis zu 1 MB/sec Durchsatz oder bis zu 10 gleichzeitige Aufrufe verarbeiten. Bei der Verwendung des Bereitstellungsmodus fallen zusätzliche Kosten an, die von Ihrer Nutzung des Event-Pollers abhängen. Details zu den Preisen finden Sie unter AWS Lambda -Preise.

Der Bereitstellungsmodus ist für Amazon MSK, selbstverwaltete Apache Kafka- und Amazon SQS SQS-Ereignisquellen verfügbar. Während Sie mit den Gleichzeitigkeitseinstellungen die Skalierung Ihrer Funktion steuern können, haben Sie im Bereitstellungsmodus die Kontrolle über den Durchsatz Ihrer Zuordnung von Ereignisquellen. Um eine maximale Leistung zu gewährleisten, müssen Sie möglicherweise beide Einstellungen unabhängig voneinander anpassen.

Der Bereitstellungsmodus ist ideal für Echtzeitanwendungen, die eine konstante Latenz bei der Ereignisverarbeitung erfordern, z. B. Finanzdienstleister, die Marktdatenfeeds verarbeiten, E-Commerce-Plattformen, die personalisierte Empfehlungen in Echtzeit bereitstellen, und Spieleunternehmen, die Live-Spielerinteraktionen verwalten.

Jeder Event Poller unterstützt unterschiedliche Durchsatzkapazitäten:

- Für Amazon MSK und selbstverwalteten Apache Kafka: bis zu 5% Durchsatz oder bis zu 5 MB/sec gleichzeitige Aufrufe
- Für Amazon SQS: bis zu 1 MB/sec Durchsatz oder bis zu 10 gleichzeitige Aufrufe oder bis zu 10 SQS-Polling-API-Aufrufe pro Sekunde.

Für Amazon SQS SQS-Ereignisquellenzuordnungen können Sie die Mindestanzahl von Pollern zwischen 2 und 200 mit einer Standardeinstellung von 2 und die maximale Anzahl zwischen 2 und 10.000 mit einer Standardeinstellung von 200 festlegen. Lambda skaliert die Anzahl der Event-Poller zwischen Ihrem konfigurierten Minimum und Maximum und fügt schnell bis zu 1.000 Parallelität pro Minute hinzu, um eine konsistente Verarbeitung Ihrer Ereignisse mit niedriger Latenz zu gewährleisten.

Für Kafka-Ereignisquellenzuordnungen können Sie die Mindestanzahl von Pollern zwischen 1 und 200 mit dem Standardwert 1 und die Höchstanzahl zwischen 1 und 2.000 mit einem Standardwert von 200 festlegen. Lambda skaliert die Anzahl der Event-Poller auf der Grundlage Ihres Event-Backlogs in Ihrem Thema zwischen Ihrem konfigurierten Minimum und Maximum, um eine Verarbeitung Ihrer Ereignisse mit geringer Latenz zu ermöglichen.

Beachten Sie, dass für Amazon SQS SQS-Ereignisquellen die Einstellung für maximale Parallelität nicht im Bereitstellungsmodus verwendet werden kann. Wenn Sie den Bereitstellungsmodus verwenden, steuern Sie die Parallelität über die Einstellung „Maximale Anzahl an Event-Pollers“.

Weitere Informationen zur Konfiguration des Bereitstellungsmodus finden Sie in folgenden Abschnitten:

- Konfiguration des Bereitstellungsmodus für Amazon MSK-Ereignisquellenzuordnungen
- Konfiguration des Bereitstellungsmodus für selbstverwaltete Apache-Kafka-Ereignisquellen-Zuordnungen
- Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen

Um die Latenz im Bereitstellungsmodus zu minimieren, setzen Sie den Wert auf 0. `MaximumBatchingWindowInSeconds` Diese Einstellung stellt sicher, dass Lambda unmittelbar nach Abschluss des aktuellen Funktionsaufrufs mit der Verarbeitung des nächsten Batches beginnt. Weitere Informationen zur Verarbeitung mit niedriger Latenz finden Sie unter Niedrige Latenz für Apache-Kafka.

Nachdem Sie den Bereitstellungsmodus konfiguriert haben, können Sie die Verwendung von Event-Pollern für Ihren Workload beobachten, indem Sie die `ProvisionedPollers`-Metrik überwachen. Weitere Informationen finden Sie unter the section called “Metriken zur Zuordnung von Ereignisquellen”.

API für die Ereignisquellenzuordnung

Um eine Ereignisquelle mit der AWS Command Line Interface (AWS CLI) oder einem AWS -SDK zu verwalten, können Sie die folgenden API-Operationen verwenden:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping

- DeleteEventSourceMapping

Verwendung von Tags für Zuordnungen von Ereignisquellen

Sie können Zuordnungen von Ereignisquellen taggen, um Ihre Ressourcen zu organisieren und zu verwalten. Tags sind frei formulierte Schlüssel-Wert-Paare, die mit Ihren Ressourcen verknüpft sind und von AWS-Services unterstützt werden. Weitere Informationen zu Anwendungsfällen für Tags finden Sie unter Allgemeine Tagging-Strategien im Tagging AWS Resources and Tag Editor Guide.

Zuordnungen von Ereignisquellen sind mit Ereignisfunktionen verknüpft, die ihre eigenen Tags haben können. Zuordnungen von Ereignisquellen erben nicht automatisch Tags von Funktionen. Sie können die AWS Lambda API verwenden, um Tags anzuzeigen und zu aktualisieren. Sie können auch Tags anzeigen und aktualisieren, während Sie eine bestimmte Ereignisquellenzuordnung in der Lambda-Konsole verwalten, einschließlich solcher, die den Bereitstellungsmodus für Amazon SQS verwenden.

Erforderliche Berechtigungen zum Arbeiten mit Tags

Um einer AWS Identity and Access Management -(IAM)-Identität (Benutzer, Gruppe oder Rolle) das Lesen oder Setzen von Tags auf einer Ressource zu ermöglichen, gewähren Sie ihr die entsprechenden Berechtigungen:

- lambda: ListTags —Wenn eine Ressource Tags enthält, gewähren Sie diese Berechtigung jedem, der sie aufrufen muss. `ListTags` Für getaggte Event-Funktionen ist diese Berechtigung auch für erforderlich. `GetFunction`
- lambda: TagResource —Erteilen Sie diese Berechtigung jedem, der bei create aufrufen `TagResource` oder ein Tag ausführen muss.

Optional können Sie erwägen, auch die lambda: UntagResource -Berechtigung zu erteilen, um `UntagResource` Aufrufe der Ressource zuzulassen.

Weitere Informationen finden Sie unter Identity-based IAM-Richtlinien für Lambda.

Verwendung von Tags mit der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um Ereignisquellenzuordnungen mit Tags zu erstellen, Tags zu vorhandenen Ereignisquellenzuordnungen hinzuzufügen und Ereignisquellenzuordnungen nach Tag zu filtern, einschließlich der im Bereitstellungsmodus für Amazon SQS konfigurierten.

Wenn Sie einen Auslöser für unterstützte Stream- und Warteschlangen-basierte Dienste über die Lambda-Konsole hinzufügen, erstellt Lambda automatisch eine Zuordnung von Ereignisquellen. Weitere Informationen zu diesen Ereignisquellen finden Sie unter the section called “Zuweisung von Ereignisquellen”. Um eine Ereignisquellenzuordnung in der Konsole zu erstellen, benötigen Sie die folgenden Voraussetzungen:

- Eine Ereignisfunktion.
- Eine Ereignisquelle von einem betroffenen Dienst.

Sie können die Tags als Teil derselben Benutzeroberfläche hinzufügen, die Sie zum Erstellen oder Aktualisieren von Triggern verwenden.

So fügen Sie ein Tag hinzu, wenn Sie eine Zuordnung von Ereignisquellen erstellen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie unter Auslöserkonfiguration in der Dropdownliste den Namen des Dienstes aus, aus dem Ihre Ereignisquelle stammt.
5. Geben Sie die Kernkonfiguration für Ihre Ereignisquelle an. Weitere Informationen zur Konfiguration Ihrer Ereignisquelle finden Sie im Abschnitt für den entsprechenden Dienst unter the section called “Integration anderer Services”.
6. Wählen Sie unter Konfiguration der Zuordnung von Ereignisquellen die Option Zusätzliche Einstellungen aus.
7. Wählen Sie unter Tags die Option Neuen Tag hinzufügen aus
8. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Einschränkungen beim Taggen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Leitfaden AWS Tagging-Ressourcen und Tag-Editor.
9. Wählen Sie Hinzufügen aus.

So fügen Sie Tags zu einer bestehenden Zuordnung von Ereignisquellen hinzu

1. Öffnen Sie die Zuordnungen von Ereignisquellen in der Lambda-Konsole.

2. Wählen Sie aus der Ressourcenliste die UUID für die Zuordnung von Ereignisquellen aus, die Ihrem Funktions- und Ereignisquellen-ARN entspricht.
3. Wählen Sie in der Registerkartenliste unter dem Konfigurationsbereich Allgemein die Option Tags.
4. Wählen Sie Tags verwalten aus.
5. Wählen Sie Neues Tag hinzufügen aus.
6. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Einschränkungen beim Taggen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Leitfaden für AWS Tag-Ressourcen und Tag-Editor.
7. Wählen Sie Speichern.

So filtern Sie Zuordnungen von Ereignisquellen nach Tag

1. Öffnen Sie die Zuordnungen von Ereignisquellen in der Lambda-Konsole.
2. Wählen Sie das Suchfeld.
3. Wählen Sie in der Dropdown-Liste Ihren Tagschlüssel unter der Tag-Unterüberschrift aus.
4. Wählen Sie Verwenden: „Tag-Name“, um alle Zuordnungen von Ereignisquellen zu sehen, die mit dieser Taste gekennzeichnet sind oder wählen Sie einen Operator, um weiter nach Werten zu filtern.
5. Wählen Sie Ihren Tag-Wert aus, um nach einer Kombination aus Tag-Schlüssel und -Wert zu filtern.

Das Suchfeld unterstützt auch die Suche nach Tag-Schlüsseln. Geben Sie den Namen eines Schlüssels ein, um ihn in der Liste zu finden.

Verwenden von Tags mit dem AWS CLI

Mit der Lambda-API können Sie Tags zu vorhandenen Lambda-Ressourcen, einschließlich Zuordnungen von Ereignisquellen, hinzufügen und entfernen. Sie können beim Erstellen einer Ereignisquellenzuordnung auch Tags hinzufügen, um eine Ressource während ihres gesamten Lebenszyklus mit Tags zu versehen.

Aktualisierung von Tags mit den Lambda-Tag-APIs

Sie können Tags für unterstützte Lambda-Ressourcen über die TagResourceund UntagResourceAPI-Operationen hinzufügen und entfernen.

Sie können diese Operationen mit der Taste AWS CLI aufrufen. Um Tags einer vorhandenen Ressource hinzuzufügen, verwenden Sie den folgenden `tag-resource`-Befehl. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Mit dem Befehl `untag-resource` können Sie Tags entfernen. In diesem Beispiel wird das Tag mit dem Schlüssel entfernt*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Hinzufügen von Tags beim Erstellen einer Zuordnung von Ereignisquellen

Verwenden Sie die CreateEventSourceMappingAPI-Operation, um eine neue Lambda-Ereignisquellenzuordnung mit Tags zu erstellen. Geben Sie den Parameter `Tags` an. Sie können diesen Vorgang mit dem `create-event-source-mapping` AWS CLI Befehl und der `--tags` Option aufrufen. Weitere Informationen über diesen CLI-Befehl finden Sie unter create-event-source-mapping in der AWS CLI -Befehlsreferenz.

Vergewissern Sie sich vor der Verwendung des `Tags`-Parameters mit `CreateEventSourceMapping`, dass Ihre Rolle neben den üblichen Berechtigungen, die für diesen Vorgang erforderlich sind, auch die Berechtigung hat, Ressourcen zu taggen. Weitere Informationen zu den Berechtigungen für das Tagging finden Sie unter the section called “Erforderliche Berechtigungen zum Arbeiten mit Tags”.

Anzeigen von Tags mit den Lambda-Tag-APIs

Um die Tags anzuzeigen, die auf eine bestimmte Lambda-Ressource angewendet werden, verwenden Sie die `ListTags`-API-Operation. Weitere Informationen finden Sie unter ListTags.

Sie können diesen Vorgang mit dem `list-tags` AWS CLI Befehl aufrufen, indem Sie einen ARN (Amazon Resource Name) angeben.

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier
```

Filtern von Ressourcen nach Tag

Sie können den AWS Resource Groups Tagging API GetResourcesAPI-Vorgang verwenden, um Ihre Ressourcen nach Tags zu filtern. Die `GetResources`-Operation empfängt bis zu 10 Filter, wobei jeder Filter einen Tag-Schlüssel und bis zu 10 Tag-Werte enthält. Sie stellen `GetResources` mit einem `ResourceType` zur Verfügung, um nach bestimmten Ressourcentypen zu filtern.

Sie können diesen Vorgang mit dem `get-resources` AWS CLI Befehl aufrufen. Beispiele für die Verwendung von `get-resources` finden Sie unter get-resources in der AWS -CLI-Befehlsreferenz.

Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Sie können zum Beispiel einen Filter hinzufügen, damit Ihre Funktion nur Amazon-SQS-Nachrichten verarbeitet, die bestimmte Datenparameter enthalten. Die Ereignisfilterung funktioniert nur mit der bestimmten Zuordnung von Ereignisquellen. Sie können den Zuordnungen von Ereignisquellen Filter für Folgendes hinzufügen: AWS-Services

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon-Simple-Queue-Service (Amazon SQS)

Spezifische Informationen zum Filtern mit bestimmten Ereignisquellen finden Sie unter the section called “Verwenden von Filtern mit unterschiedlichen AWS-Services”. Lambda unterstützt keine Ereignisfilterung für Amazon DocumentDB.

Standardmäßig können Sie bis zu fünf verschiedene Filter für eine einzelne Zuordnung von Ereignisquellen definieren. Ihre Filter sind logisch via OR verknüpft. Wenn ein Datensatz aus Ihrer Ereignisquelle einen oder mehrere Ihrer Filter erfüllt, nimmt Lambda den Datensatz in das nächste Ereignis auf, das es an Ihre Funktion sendet. Wenn keiner Ihrer Filter erfüllt ist, verwirft Lambda den Datensatz.

> **Note**
>
> Wenn Sie mehr als fünf Filter für eine Ereignisquelle definieren müssen, können Sie eine Kontingenterhöhung auf bis zu 10 Filter für jede Ereignisquelle beantragen. Wenn Sie versuchen, mehr Filter hinzuzufügen, als Ihr aktuelles Kontingent zulässt, gibt Lambda beim Versuch, die Ereignisquelle zu erstellen, einen Fehler zurück.

Themen

- Grundlagen der Ereignisfilterung verstehen

- Umgang mit Datensätzen, die die Filterkriterien nicht erfüllen
- Filterregelsyntax
- Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole)
- Hinzufügen von Filterkriterien zu einer Ereignisquellenzuordnung (AWS CLI)
- Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS SAM)
- Verschlüsselung der Filterkriterien
- Verwenden von Filtern mit unterschiedlichen AWS-Services

Grundlagen der Ereignisfilterung verstehen

Ein Filterkriterienobjekt (`FilterCriteria`) ist eine Struktur, die aus einer Liste von Filtern (`Filters`) besteht. Jeder Filter ist eine Struktur, die ein Ereignisfiltermuster (`Pattern`) definiert. Ein Muster ist eine Zeichenkette, die eine JSON-Filterregel darstellt. Die Struktur eines `FilterCriteria`-Objekts ist wie folgt.

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

Ihr Filtermuster kann Metadateneigenschaften, Dateneigenschaften oder beides enthalten. Die verfügbaren Metadaten-Parameter und das Format der Daten-Parameter variieren je nach dem AWS-Service , der als Ereignisquelle fungiert. Nehmen wir zum Beispiel an, dass Ihre Zuordnung von Ereignisquellen den folgenden Datensatz von einer Amazon-SQS-Warteschlange empfängt:

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\\n \"City\": \"Seattle\",\\n \"State\": \"WA\",\\n \"Temperature\":
 \"46\"\\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- Metadateneigenschaften sind die Felder, die Informationen über das Ereignis enthalten, das den Datensatz erstellt hat. Im Beispiel für den Amazon-SQS-Datensatz umfassen die Metadateneigenschaften Felder wie `messageID`, `eventSourceArn` und `awsRegion`.
- Dateneigenschaften sind die Felder des Datensatzes, die die Daten aus Ihrem Stream oder Ihrer Warteschlange enthalten. Im Amazon-SQS-Ereignisbeispiel ist der Schlüssel für das Datenfeld `body`, und die Dateneigenschaften sind die Felder `City`, `State` und `Temperature`.

Verschiedene Arten von Ereignisquellen verwenden unterschiedliche Schlüsselwerte für ihre Datenfelder. Um nach Dateneigenschaften zu filtern, stellen Sie sicher, dass Sie den richtigen Schlüssel im Muster Ihres Filters verwenden. Eine Liste der Datenfilterschlüssel und Beispiele für Filtermuster für jeden unterstützten AWS-Service Schlüssel finden Sie unterVerwenden von Filtern mit unterschiedlichen AWS-Services.

Die Ereignisfilterung kann mehrstufige JSON-Filterung verarbeiten. Betrachten Sie zum Beispiel das folgende Fragment eines Datensatzes aus einem DynamoDB-Stream:

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
```

```
            "N": "1234"
    },
    ...
}
```

Angenommen, Sie möchten nur die Datensätze verarbeiten, bei denen die `Number` des Sortierschlüssels 4567 ist. In diesem Fall würde Ihr `FilterCriteria`-Objekt wie folgt aussehen:

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

Umgang mit Datensätzen, die die Filterkriterien nicht erfüllen

Die Art und Weise, wie Lambda mit Datensätzen umgeht, die Ihren Filterkriterien nicht entsprechen, hängt von der Ereignisquelle ab.

- Bei Amazon SQS entfernt Lambda die Nachricht automatisch aus der Warteschlange, wenn die Nachricht Ihren Filterkriterien nicht entspricht. Sie müssen diese Nachrichten in Amazon SQS nicht manuell löschen.
- Bei Kinesis und DynamoDB schreitet der Stream-Iterator nach der Auswertung eines Datensatzes durch Ihre Filterkriterien über diesen Datensatz hinaus. Wenn der Datensatz Ihre Filterkriterien nicht erfüllt, müssen Sie den Datensatz nicht manuell aus Ihrer Ereignisquelle löschen. Nach Ablauf der Aufbewahrungsfrist löschen Kinesis und DynamoDB diese alten Datensätze automatisch.

Wenn Sie möchten, dass Datensätze früher gelöscht werden, lesen Sie Ändern des Zeitraums der Datenaufbewahrung.

- Bei Amazon MSK-, selbstverwalteten Apache-Kafka- und Amazon MQ-Nachrichten verwirft Lambda Nachrichten, die nicht allen im Filter enthaltenen Feldern entsprechen. Für Amazon MSK und den selbstverwalteten Apache Kafka schreibt Lambda Offsets für übereinstimmende und nicht zugeordnete Nachrichten fest, nachdem die Event-Funktion erfolgreich aufgerufen wurde. Bei Amazon MQ bestätigt Lambda übereinstimmende Nachrichten nach erfolgreichem Aufruf der Funktion und bestätigt nicht übereinstimmende Nachrichten beim Filtern.

Filterregelsyntax

Für Filterregeln unterstützt Lambda die EventBridge Amazon-Regeln und verwendet dieselbe Syntax wie EventBridge. Weitere Informationen finden Sie unter Amazon EventBridge Event Patterns im EventBridge Amazon-Benutzerhandbuch.

Im Folgenden finden Sie eine Zusammenfassung aller Vergleichsoperatoren, die für die Lambda-Ereignisfilterung verfügbar sind.

Vergleichsoperator	Beispiel	Regelsyntax
Null	UserID is null	"UserID": [null]
Leer	LastName ist leer	"LastName": [""]
Gleich	Name is "Alice"	"Name": ["Alice"]
Gleich (Groß-/Kleinschreibung ignorieren)	Name is "Alice"	"Name": [{ "equals-ignore-case": "alice" }]
Bedingung 2	Location is "New York" and Day is "Monday"	"Location": ["New York"], "Day": ["Monday"]
Oder	PaymentType ist „Kredit“ oder „Lastschrift“	"PaymentType„: [„Kredit“, „Lastschrift"]
Oder (mehrere Felder)	Location is "New York", or Day is "Monday".	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]

Vergleichsoperator	Beispiel	Regelsyntax
Nicht	Weather is anything but "Raining"	"Weather": [{ "anything-but": ["Raining"] }]
Numeric (equals)	Price is 100	"Price": [{ "numeric": ["=", 100] }]
Numeric (range)	Price is more than 10, and less than or equal to 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
Vorhanden	ProductName existiert	"ProductName„: [{„existiert“: wahr}]
Nicht vorhanden	ProductName existiert nicht	"ProductName„: [{„existiert“: falsch}]
Beginnt mit	Region is in the US	"Region": [{"prefix": "us-" }]
Endet mit	FileName endet mit der Erweiterung.png.	"FileName„: [{„Suffix“: „.png“}]

> **Note**
>
> Wie bei EventBridge Zeichenketten verwendet Lambda den exakten Zeichenabgleich ohne Umschaltung der Groß- und Kleinschreibung oder eine andere Normalisierung von Zeichenketten. Bei Zahlen verwendet Lambda auch eine Zeichenfolgendarstellung. 300, 300,0 und 3,0e2 werden z. B. nicht gleich behandelt.

Beachten Sie, dass der Exists-Operator nur für Blattknoten in Ihrer JSON-Ereignisquelle funktioniert. Es passt nicht zu Zwischenknoten. Bei der folgenden JSON-Datei `{ "person": { "address": [ { "exists": true } ] } }` würde das Filtermuster beispielsweise keine Übereinstimmung finden, da es sich um einen Zwischenknoten `"address"` handelt.

```
{
  "person": {
    "name": "John Doe",
```

```
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole)

Führen Sie diese Schritte aus, um eine neue Ereignisquellenzuordnung mit Filterkriterien über die Lambda-Konsole zu erstellen.

Eine neue Ereignisquellenzuordnung mit Filterkriterien erstellen (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Ereignisfunktion, für die Sie eine Ereignisquellenzuordnung erstellen möchten.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie bei Trigger configuration (Auslöserkonfiguration) einen Auslösertyp aus, der Ereignisfilterung unterstützt. Eine Liste der unterstützten Services finden Sie in der Liste am Anfang dieser Seite.
5. Erweitern Sie Additional settings (Zusätzliche Einstellungen).
6. Wählen Sie unter Filter criteria (Filterkriterien) die Option Add (Hinzufügen) aus und definieren Sie anschließend Ihre Filter. Sie können z. B. Folgendes eingeben.

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

   Damit wird Lambda angewiesen, nur Datensätze zu verarbeiten, in denen das Feld `Metadata` gleich 1 oder 2 ist. Sie können weiterhin Hinzufügen auswählen, um weitere Filter bis zur maximal zulässigen Anzahl hinzuzufügen.
7. Wenn Sie fertig mit dem Hinzufügen Ihrer Filter sind, klicken Sie auf Speichern.

Wenn Sie Filterkriterien über die Konsole eingeben, geben Sie nur das Filtermuster ein und müssen weder den `Pattern`-Schlüssel noch Escape-Anführungszeichen angeben. In Schritt 6 der Anweisungen oben entspricht `{ "Metadata" : [ 1, 2 ] }` den folgenden `FilterCriteria`.

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Nachdem Sie Ihre Ereignisquellenzuordnung in der Konsole erstellt haben, sehen Sie die formatierten `FilterCriteria` in den Auslöserdetails. Weitere Beispiele zum Erstellen von Ereignisfiltern mithilfe der -Konsole finden Sie unter Verwenden von Filtern mit unterschiedlichen AWS-Services.

Hinzufügen von Filterkriterien zu einer Ereignisquellenzuordnung (AWS CLI)

Angenommen, Sie möchten, dass eine Ereignisquellenzuordnung folgende hat `FilterCriteria`:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

Dieser create-event-source-mapping-Befehl erstellt eine neue Amazon SQS Zuordnung von Ereignisquellen für Funktion `my-function` mit den angegebenen `FilterCriteria`.

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

Beachten Sie, dass Sie zum Aktualisieren einer Ereignisquellenzuordnung ihre UUID benötigen. Sie können die UUID über einen list-event-source-mappings-Aufruf erhalten. Lambda gibt die UUID auch in der create-event-source-mapping-CLI-Antwort zurück.

Um Filterkriterien aus einer Ereignisquelle zu entfernen, können Sie den folgenden Befehl update-event-source-mapping mit einem leeren `FilterCriteria`-Objekt ausführen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

Weitere Beispiele für das Erstellen von Ereignisfiltern mithilfe von finden Sie unterVerwenden von Filtern mit unterschiedlichen AWS-Services. AWS CLI

Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS SAM)

Angenommen, Sie möchten eine Ereignisquelle so konfigurieren AWS SAM , dass sie die folgenden Filterkriterien verwendet:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Um diese Filterkriterien zu Ihrer Zuordnung von Ereignisquellen hinzuzufügen, fügen Sie das folgende Snippet in die YAML-Vorlage für Ihre Ereignisquelle ein.

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

Weitere Informationen zum Erstellen und Konfigurieren einer AWS SAM Vorlage für eine Ereignisquellenzuordnung finden Sie im EventSourceAbschnitt des AWS SAM Entwicklerhandbuchs.

Weitere Beispiele für die Erstellung von Ereignisfiltern mithilfe von AWS SAM Vorlagen finden Sie unterVerwenden von Filtern mit unterschiedlichen AWS-Services.

Verschlüsselung der Filterkriterien

Standardmäßig verschlüsselt Lambda Ihr Filterkriterienobjekt nicht. Für Anwendungsfälle, in denen Sie vertrauliche Informationen in Ihr Filterkriterienobjekt aufnehmen könnten, können Sie diese mit Ihrem eigenen KMS-Schlüssel verschlüsseln.

Nachdem Sie Ihr Filterkriterienobjekt verschlüsselt haben, können Sie seine Klartextversion mithilfe eines GetEventSourceMappingAPI-Aufrufs anzeigen. Sie müssen über `kms:Decrypt`-Berechtigungen verfügen, um die Filterkriterien im Klartext anzeigen zu können.

> **Note**
>
> Wenn Ihr Filterkriterienobjekt verschlüsselt ist, redigiert Lambda den Wert des `FilterCriteria` Felds in der ListEventSourceMappingsAntwort auf Aufrufe. Stattdessen wird dieses Feld als `null` angezeigt. Verwenden Sie die API`FilterCriteria`, um den wahren Wert von zu ermitteln. GetEventSourceMapping
> Um den entschlüsselten Wert von `FilterCriteria` in der Konsole anzuzeigen, stellen Sie sicher, dass Ihre IAM-Rolle Berechtigungen für enthält. GetEventSourceMapping

Sie können Ihren eigenen KMS-Schlüssel über die Konsole API/CLI, oder angeben. CloudFormation

Um Filterkriterien mit einem kundeneigenen KMS-Schlüssel zu verschlüsseln (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Add trigger. Wenn Sie bereits über einen vorhandenen Trigger verfügen, wählen Sie die Registerkarte Konfiguration und dann Trigger aus. Wählen Sie den vorhandenen Auslöser aus und klicken Sie auf Edit (Bearbeiten).
3. Aktivieren Sie das Kontrollkästchen neben Mit dem vom Kunden verwalteten KMS-Schlüssel verschlüsseln.
4. Wählen Sie für Wählen Sie einen vom Kunden verwalteten KMS-Verschlüsselungsschlüssel einen vorhandenen aktivierten Schlüssel aus oder erstellen Sie einen neuen Schlüssel. Je nach Vorgang benötigen Sie einige oder alle der folgenden Berechtigungen: `kms:DescribeKey`, `kms:GenerateDataKey` und `kms:Decrypt`. Verwenden Sie die KMS-Schlüsselrichtlinie, um diese Berechtigungen zu gewähren.

Wenn Sie Ihren eigenen KMS-Schlüssel verwenden, müssen die folgenden API-Vorgänge in der Schlüsselrichtlinie zugelassen sein:

- `kms:Decrypt` – Muss dem regionalen Lambda-Serviceprinzipal gewährt werden (`lambda.AWS_region.amazonaws.com`). Dies ermöglicht Lambda die Entschlüsselung von Daten mit diesem KMS-Schlüssel.
  - Um ein dienstübergreifendes Problem mit verwirrten Stellvertretern zu vermeiden, verwendet die Schlüsselrichtlinie den `aws:SourceArn` globalen Bedingungsschlüssel. Der richtige Wert des `aws:SourceArn`-Schlüssels ist der ARN Ihrer Ressource für die Zuordnung von Ereignisquellen. Sie können ihn also erst dann zu Ihrer Richtlinie hinzufügen, wenn Sie den ARN kennen. Lambda leitet auch die `aws:lambda:FunctionArn`- und `aws:lambda:EventSourceArn`-Schlüssel und ihre jeweiligen Werte im Verschlüsselungskontext weiter, wenn eine Entschlüsselungsanforderung an KMS gestellt wird. Diese Werte müssen den in der Schlüsselrichtlinie angegebenen Bedingungen entsprechen, damit die Entschlüsselungsanforderung erfolgreich ist. Sie müssen EventSourceArn für Self-managed Kafka keine Ereignisquellen angeben, da sie keine haben. EventSourceArn
- `kms:Decrypt`— Muss auch dem Prinzipal gewährt werden, der beabsichtigt, den Schlüssel zur Anzeige der Klartext-Filterkriterien in GetEventSourceMappingunseren DeleteEventSourceMappingAPI-Aufrufen zu verwenden.
- `kms:DescribeKey` – Stellt die vom Kunden verwalteten Schlüsseldetails bereit, um dem angegebenen Prinzipal die Verwendung des Schlüssels zu ermöglichen.
- `kms:GenerateDataKey` – Ermöglicht Lambda die Generierung eines Datenschlüssels zur Verschlüsselung der Filterkriterien im Namen des angegebenen Prinzipals (Umschlagverschlüsselung).

Sie können AWS CloudTrail damit AWS KMS Anfragen verfolgen, die Lambda in Ihrem Namen stellt. CloudTrail Beispielereignisse finden Sie unter???.

Wir empfehlen außerdem, den `kms:ViaService`-Bedingungsschlüssel zu verwenden, um die Verwendung des KMS-Schlüssels nur auf Anfragen von Lambda zu beschränken. Der Wert dieses Schlüssels ist der regionale Lambda-Serviceprinzipal (`lambda.AWS_region.amazonaws.com`). Im Folgenden finden Sie ein Beispiel für eine Schlüsselrichtlinie, die alle relevanten Berechtigungen gewährt:

Example AWS KMS wichtige Richtlinie

JSON

```
{
    "Version":"2012-10-17",
    "Id": "example-key-policy-1",
    "Statement": [
        {
            "Sid": "Allow Lambda to decrypt using the key",
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.us-east-1.amazonaws.com"
            },
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals" : {
                    "aws:SourceArn": [
                        "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:<esm_uuid>"
                    ]
                },
                "StringEquals": {

 "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:test-function",

 "kms:EncryptionContext:aws:lambda:EventSourceArn": "arn:aws:sqs:us-
east-1:123456789012:test-queue"
                }
            }
        },
        {
            "Sid": "Allow actions by an AWS account on the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:root"
            },
            "Action": "kms:*",
            "Resource": "*"
```

```
        },
        {
            "Sid": "Allow use of the key to specific roles",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/ExampleRole"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals" : {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        }
    ]
}
```

Um Ihren eigenen KMS-Schlüssel zum Verschlüsseln von Filterkriterien zu verwenden, können Sie auch den folgenden CreateEventSourceMapping AWS CLI Befehl verwenden. Geben Sie den KMS-Schlüssel ARN mit dem `--kms-key-arn`-Flag an.

```
aws lambda create-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

Wenn Sie bereits über eine Ereignisquellenzuordnung verfügen, verwenden Sie stattdessen den UpdateEventSourceMapping AWS CLI Befehl. Geben Sie den KMS-Schlüssel ARN mit dem `--kms-key-arn`-Flag an.

```
aws lambda update-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
```

```
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

Dieser Vorgang überschreibt alle zuvor angegebenen KMS-Schlüssel. Wenn Sie das `--kms-key-arn`-Flag zusammen mit einem leeren Argument angeben, verwendet Lambda Ihren KMS-Schlüssel nicht mehr zum Verschlüsseln von Filterkriterien. Stattdessen verwendet Lambda standardmäßig wieder einen Amazon-owned Schlüssel.

Um Ihren eigenen KMS-Schlüssel in einer CloudFormation Vorlage anzugeben, verwenden Sie die `KMSKeyArn` Eigenschaft des `AWS::Lambda::EventSourceMapping` Ressourcentyps. Sie können beispielsweise das folgende Snippet in die YAML-Vorlage für Ihre Ereignisquelle einfügen.

```
MyEventSourceMapping:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    ...
    FilterCriteria:
      Filters:
        - Pattern: '{"a": [1, 2]}'
    KMSKeyArn: "arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599"
    ...
```

Um Ihre verschlüsselten Filterkriterien in einem GetEventSourceMappingoder DeleteEventSourceMappingAPI-Aufruf im Klartext anzeigen zu können, benötigen Sie entsprechende `kms:Decrypt` Berechtigungen.

Ab dem 6. August 2024 wird das `FilterCriteria` Feld nicht mehr in AWS CloudTrail Protokollen von CreateEventSourceMapping, und DeleteEventSourceMappingAPI-Aufrufen angezeigt UpdateEventSourceMapping, wenn Ihre Funktion keine Ereignisfilterung verwendet. Wenn Ihre Funktion die Ereignisfilterung verwendet, wird das `FilterCriteria`-Feld als leer (`{}`) angezeigt. Sie können Ihre Filterkriterien in der Antwort auf GetEventSourceMappingAPI-Aufrufe weiterhin im Klartext anzeigen, wenn Sie über `kms:Decrypt` Berechtigungen für den richtigen KMS-Schlüssel verfügen.

Beispiel für einen CloudTrail Protokolleintrag für Anrufe Create/Update/DeleteEventSourceMapping

Im folgenden AWS CloudTrail Beispiel wird ein Protokolleintrag für einen CreateEventSourceMapping Anruf als leer (`{}`) `FilterCriteria` angezeigt, da die Funktion eine Ereignisfilterung verwendet. Dies ist auch dann der Fall, wenn `FilterCriteria`-Objekt gültige Filterkriterien enthält, die Ihre

Funktion aktiv verwendet. Wenn die Funktion keine Ereignisfilterung verwendet, CloudTrail wird das `FilterCriteria` Feld in Protokolleinträgen überhaupt nicht angezeigt.


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:userid1",
        "arn": "arn:aws:sts::123456789012:assumed-role/Example/example-role",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA987654321EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/User1",
                "accountId": "123456789012",
                "userName": "User1"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-05-09T20:35:01Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2024-05-09T21:05:41Z",
    "eventSource": "lambda.amazonaws.com",
    "eventName": "CreateEventSourceMapping20150331",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "AWS Internal",
    "userAgent": "AWS Internal",
    "requestParameters": {
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "functionName": "example-function",
        "enabled": true,
        "batchSize": 10,
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "scalingConfig": {},
        "maximumBatchingWindowInSeconds": 0,
```

```
        "sourceAccessConfigurations": []
    },
    "responseElements": {
        "uUID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
        "batchSize": 10,
        "maximumBatchingWindowInSeconds": 0,
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:example-
function",
        "lastModified": "May 9, 2024, 9:05:41 PM",
        "state": "Creating",
        "stateTransitionReason": "USER_INITIATED",
        "functionResponseTypes": [],
        "eventSourceMappingArn": "arn:aws:lambda:us-east-2:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```


Verwenden von Filtern mit unterschiedlichen AWS-Services

Verschiedene Arten von Ereignisquellen verwenden unterschiedliche Schlüsselwerte für ihre Datenfelder. Um nach Dateneigenschaften zu filtern, stellen Sie sicher, dass Sie den richtigen Schlüssel im Muster Ihres Filters verwenden. In der folgenden Tabelle sind die Filterschlüssel für jeden unterstützten Schlüssel aufgeführt AWS-Service.

AWS-Service	Filterschlüssel
DynamoDB	`dynamodb`
Kinesis	`data`

AWS-Service	Filterschlüssel
Amazon MQ	`data`
Amazon MSK	`value`
Self-managed Apache Kafka	`value`
Amazon SQS	`body`

In den folgenden Abschnitten finden Sie Beispiele für Filtermuster für verschiedene Arten von Ereignisquellen. Sie enthalten auch Definitionen der unterstützten Formate für eingehende Daten und Filtermuster für jeden unterstützten Service.

- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”

Testen von Lambda-Ereignisfunktionen in der Konsole

Sie können Ihre Lambda-Ereignisfunktionen in der Konsole testen, indem Sie Ihre Funktion mit einem Testereignis aufrufen. Ein Testereignis ist eine JSON-Eingabe für Ihre Event-Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres Dokument (`{}`) sein.

Wenn Sie einen Test in der Konsole ausführen, ruft Lambda Ihre Funktion synchron mit dem Testereignis auf. Die Funktionslaufzeit konvertiert das Ereignis-JSON in ein Objekt und übergibt es zur Verarbeitung an die Handler-Methode Ihres Codes.

> **Erstellen Sie ein Testereignis**
>
> Bevor Sie in der Konsole testen können, müssen Sie ein privates oder gemeinsam nutzbares Testereignis erstellen.

Aufrufen von Ereignisfunktionen mit Testereignissen

Um eine Ereignisfunktion zu testen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Event-Funktion, die Sie testen möchten.
3. Wählen Sie die Registerkarte Test.
4. Wählen Sie unter Test event (Testereignis) die Option Create new event (Neues Ereignis erstellen) oder Edit saved event (Gespeichertes Ereignis bearbeiten) und dann das gespeicherte Ereignis aus, das Sie verwenden möchten.
5. Optional — wählen Sie ein Template (Vorlage) für den Event-JSON.
6. Wählen Sie Test aus.
7. Erweitern Sie unter Execution result (Ausführungsergebnis) die Option Details, um die Testergebnisse anzuzeigen.

Um Ihre Ereignisfunktion aufzurufen, ohne Ihr Testereignis zu speichern, wählen Sie Vor dem Speichern testen. Dadurch wird ein nicht gespeichertes Testereignis erstellt, das Lambda nur für die Dauer der Sitzung beibehält.

Für die Node.js Laufzeiten Python und Ruby können Sie auch auf der Registerkarte Code auf Ihre vorhandenen gespeicherten und nicht gespeicherten Testereignisse zugreifen. Verwenden Sie den Abschnitt TESTEREIGNISSE, um Tests zu erstellen, zu bearbeiten und auszuführen.

Private Testereignisse erstellen

Private Testereignisse stehen nur dem Ereignisersteller zur Verfügung und benötigen keine zusätzlichen Berechtigungen zur Verwendung. Sie können bis zu 10 private Testereignisse pro Event-Funktion erstellen und speichern.

So erstellen Sie ein privates Testereignis

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Event-Funktion, die Sie testen möchten.
3. Wählen Sie die Registerkarte Test.
4. Erledigen Sie unter Testereignis Folgendes:
    a. Wählen Sie eine Vorlage.
    b. Geben Sie einen Namen für den Test an.
    c. Geben Sie im Texteingabefeld das JSON-Testereignis ein.
    d. Unter Ereignisfreigabeeinstellungen wählen Sie Privat aus.
5. Wählen Sie Änderungen speichern aus.

Für die Node.js Laufzeiten Python und Ruby können Sie auch Testereignisse auf der Registerkarte Code erstellen. Verwenden Sie den Abschnitt TESTEREIGNISSE, um Tests zu erstellen, zu bearbeiten und auszuführen.

Freigabefähige Testereignisse erstellen

Gemeinsam nutzbare Testereignisse sind Testereignisse, die Sie mit anderen Benutzern desselben AWS Kontos teilen können. Sie können gemeinsam nutzbare Testereignisse anderer Benutzer bearbeiten und Ihre Event-Funktion mit ihnen aufrufen.

Lambda speichert gemeinsam nutzbare Testereignisse als Schemas in einer Amazon EventBridge (CloudWatch Events) -Schemaregistry mit dem Namen. `lambda-testevent-schemas` Da Lambda diese Registrierung verwendet, um gemeinsam nutzbare Testereignisse, die Sie erstellen, zu speichern und aufzurufen, empfehlen wir, diese Registrierung nicht zu bearbeiten oder eine Registrierung mit dem `lambda-testevent-schemas` Namen zu erstellen.

Um gemeinsam nutzbare Testereignisse zu sehen, zu teilen und zu bearbeiten, benötigen Sie Berechtigungen für alle der folgenden API-Operationen EventBridge (CloudWatch Ereignisse) für die Schemaregistrierung:

- Schemas. CreateRegistry
- Schemas. CreateSchema
- Schemas. DeleteSchema
- Schemas. DeleteSchemaVersion
- Schemas. DescribeRegistry
- Schemas. DescribeSchema
- Schemas. GetDiscoveredSchema
- Schemas. ListSchemaVersions
- Schemas. UpdateSchema

Beachten Sie, dass das Speichern von Änderungen an einem freigabefähigen Testereignis dieses Ereignis überschreibt.

Wenn Sie keine freigabefähigen Testereignisse erstellen, bearbeiten oder anzeigen können, überprüfen Sie, ob Ihr Konto über die erforderlichen Berechtigungen für diese Operationen verfügt. Wenn Sie über die erforderlichen Berechtigungen verfügen, aber immer noch nicht auf gemeinsam nutzbare Testereignisse zugreifen können, suchen Sie nach ressourcenbasierten Richtlinien, die den Zugriff auf die Registrierung EventBridge (CloudWatch Ereignisse) einschränken könnten.

So erstellen Sie ein freigabefähiges Testereignis

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Event-Funktion, die Sie testen möchten.
3. Wählen Sie die Registerkarte Test.
4. Erledigen Sie unter Testereignis Folgendes:
    a. Wählen Sie eine Vorlage.
    b. Geben Sie einen Namen für den Test an.
    c. Geben Sie im Texteingabefeld das JSON-Testereignis ein.
    d. Wählen Sie unter Event sharing settings (Ereignisfreigabeeinstellungen) Shareable (Freigabefähig) aus.

5. Wählen Sie Änderungen speichern aus.

> Verwenden Sie gemeinsam nutzbare Testereignisse mit AWS Serverless Application Model.
>
> Sie können es verwenden AWS SAM , um gemeinsam nutzbare Testereignisse aufzurufen. Weitere Informationen finden Sie im Developer GuideAWS Serverless Application Model

Löschen von freigabefähigen Test-Ereignisschemas

Wenn Sie freigabefähige Testereignisse löschen, entfernt Lambda sie aus der `lambda-testevent-schemas`-Registry. Wenn Sie das letzte freigabefähige Testereignis aus der Registry entfernen, löscht Lambda die Registry.

Wenn Sie die Event-Funktion löschen, löscht Lambda keine zugehörigen gemeinsam nutzbaren Testereignisschemas. Sie müssen diese Ressourcen manuell über die Konsole EventBridge (CloudWatch Ereignisse) bereinigen.

Status der Lambda-Ereignisfunktion

Lambda enthält in der Funktionskonfiguration für alle Eventfunktionen ein State-Feld, um anzuzeigen, wann Ihre Funktion zum Aufrufen bereit ist. `State`liefert Informationen über den aktuellen Status der Funktion, einschließlich der Frage, ob Sie die Funktion erfolgreich aufrufen können. Status von Ereignisfunktionen ändern weder das Verhalten von Funktionsaufrufen noch die Art und Weise, wie Ihre Funktion den Code ausführt.

> **Note**
>
> Die Statusdefinitionen von Ereignisfunktionen unterscheiden sich je nach Funktion geringfügig. SnapStart Weitere Informationen finden Sie unter Status von Lambda SnapStart - und Eventfunktionen.

In vielen Fällen ist eine DynamoDB-Tabelle eine ideale Methode zum Bewahren des Status zwischen Aufrufen, da sie Datenzugriff mit geringer Latenz ermöglicht und mit dem Lambda-Service skaliert werden kann. Sie können Daten auch in Amazon EFS for Lambda speichern, wenn Sie diesen Service verwenden. Dadurch wird ein Zugriff mit geringer Latenz auf den Dateisystemspeicher ermöglicht.

Zu den Zuständen von Ereignisfunktionen gehören:

- `Pending` – Nachdem Lambda die Funktion erstellt hat, setzt es den Status auf „Ausstehend“. Im Status „Ausstehend“ versucht Lambda, Ressourcen für die Funktion zu erstellen oder zu konfigurieren, beispielsweise VPC- oder EFS-Ressourcen. Lambda ruft während des ausstehenden Status keine Funktion auf. Alle Aufrufe oder andere API-Aktionen, die mit der Funktion ausgeführt werden, schlagen fehl.
- `Active` – Ihre Funktion wechselt in den aktiven Status, nachdem Lambda die Ressourcenkonfiguration und -Bereitstellung abgeschlossen hat. Event-Funktionen können nur erfolgreich aufgerufen werden, wenn sie aktiv sind.
- `Failed` – Gibt an, dass bei der Ressourcenkonfiguration oder bei der Bereitstellung ein Fehler aufgetreten ist. Wenn die Funktionserstellung fehlschlägt, setzt Lambda den Funktionsstatus auf Fehlgeschlagen, und Sie müssen die Funktion löschen und neu erstellen.
- `Inactive` – Eine Funktion wird inaktiv, wenn sie lange genug im Leerlauf war, damit Lambda die externen Ressourcen zurückfordern kann, die für sie konfiguriert wurden. Wenn Sie versuchen, eine Funktion aufzurufen, die inaktiv ist, schlägt der Aufruf fehl und Lambda setzt die Funktion

auf den Status „Ausstehend“, bis die Funktionsressourcen neu erstellt werden. Wenn Lambda die Ressourcen nicht neu erstellen kann, kehrt die Funktion in den inaktiven Zustand zurück. Möglicherweise müssen Sie alle Fehler beheben und Ihre Funktion erneut bereitstellen, um sie wieder in den aktiven Zustand zu versetzen.

Eine Funktion wechselt auch in den `Inactive` Status, wenn Lambda den Zugriff auf das Quellartefakt der Funktion verliert. Dies gilt für Container-Image-Funktionen, bei denen das Bild aus Amazon ECR gelöscht oder Berechtigungen widerrufen werden, und für .zip-Funktionen, die selbstverwalteten S3-Codespeicher verwenden, bei denen das Quellobjekt gelöscht oder Bucket-Policy-Berechtigungen widerrufen werden. Um die Funktion wiederherzustellen, stellen Sie den Zugriff auf das Quellartefakt wieder her und aktualisieren Sie die Funktion.

Wenn Sie SDK-based Automatisierungsworkflows verwenden oder die Service-APIs von Lambda direkt aufrufen, stellen Sie sicher, dass Sie vor dem Aufruf den Status einer Funktion überprüfen, um sicherzustellen, dass sie aktiv ist. Sie können dies mit der Lambda-API-Aktion GetFunctionoder durch die Konfiguration eines Kellners mithilfe des AWS SDK for Java 2.0 tun.

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

Die Ausgabe sollte folgendermaßen aussehen:

```
[
 "Active",
 "Successful"
]
```

Die folgenden Operationen schlagen fehl, während die Funktionserstellung aussteht:

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Status der Ereignisfunktion bei Aktualisierungen

Lambda hat zwei Operationen zum Aktualisieren von Ereignisfunktionen:

- UpdateFunctionCode: Aktualisiert das Bereitstellungspaket der Funktion
- UpdateFunctionConfiguration: Aktualisiert die Konfiguration der Funktion

Lambda verwendet das LastUpdateStatusAttribut, um den Fortschritt dieser Aktualisierungsvorgänge zu verfolgen. Während ein Update läuft (wenn `"LastUpdateStatus": "InProgress"`):

- Der Status der Funktion bleibt `Active`.
- Bei Aufrufen werden weiterhin der vorherige Code und die vorherige Konfiguration der Funktion verwendet, bis die Aktualisierung abgeschlossen ist.
- Der folgende Vorgang schlägt fehl:
  - UpdateFunctionCode
  - UpdateFunctionConfiguration
  - PublishVersion
  - TagResource

Wenn ein Update fehlschlägt (wann`"LastUpdateStatus": "Failed"`):

- Der Status der Funktion bleibt `Active`.
- Aufrufe verwenden weiterhin den vorherigen Code und die vorherige Konfiguration der Funktion.

Example GetFunctionConfiguration Antwort

Das folgende Beispiel ist das Ergebnis einer GetFunctionConfigurationAnfrage für eine Funktion, die gerade aktualisiert wird.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
```

```
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    ...
}
```

Grundlegendes zum Wiederholungsverhalten in Lambda-Ereignisfunktionen

Wenn Sie eine Ereignisfunktion direkt aufrufen, legen Sie die Strategie für die Behandlung von Fehlern im Zusammenhang mit Ihrem Funktionscode fest. Lambda wiederholt diese Art von Fehlern nicht automatisch in Ihrem Namen. Um den Vorgang zu wiederholen, können Sie die Funktion manuell erneut aufrufen, das fehlgeschlagene Ereignis zum Debuggen an eine Warteschlange senden oder den Fehler ignorieren. Der Code Ihrer Funktion wurde möglicherweise vollständig, teilweise oder überhaupt nicht ausgeführt. Wenn Sie es erneut versuchen, stellen Sie sicher, dass der Code Ihrer Funktion das gleiche Ereignis mehrmals verarbeiten kann, ohne doppelte Transaktionen oder andere unerwünschte Nebenwirkungen zu verursachen.

Wenn Sie eine Funktion indirekt aufrufen, müssen Sie sich über das Wiederholungsverhalten des Aufrufers und alle Services im Klaren sein, mit denen die Anforderung während des Vorgangs konfrontiert wird. Dies umfasst die folgenden Szenarien.

- Asynchroner Aufruf – Lambda startet bei Funktionsfehlern zwei Wiederholungsversuche. Wenn die Kapazität der Funktion nicht für die Verarbeitung aller eingehenden Anforderungen ausreicht, verbleiben die an die Funktion zu sendenden Ereignisse möglicherweise stundenlang in der Warteschlange. Sie können für die Funktion eine Warteschlange für unzustellbare Nachrichten konfigurieren, um nicht erfolgreich verarbeitete Ereignisse aufzufangen. Weitere Informationen finden Sie unter the section called “Dead-letter Warteschlangen”.
- Ereignisquellen-Mappings – Ereignisquellen-Mappings, die aus Streams lesen, führen für den gesamten Batch von Elementen einen Wiederholungsversuch durch. Wiederholte Fehlermeldungen blockieren die Verarbeitung des betroffenen Shards, bis der Fehler behoben ist oder die Elemente ablaufen. Zum Erkennen blockierter Shards können Sie die Metrik Iterator Age überwachen.

  Für Ereignisquellen-Zuweisungen, die aus einer Warteschlange lesen, bestimmen Sie die Zeitdauer zwischen Wiederholversuchen und dem Ziel für fehlgeschlagene Ereignisse, indem Sie für die Quellwarteschlange die Zeitbeschränkung für die Sichtbarkeit und die Redrive-Richtlinie konfigurieren. Weitere Informationen finden Sie unter Wie Lambda Datensätze aus Stream- und warteschlangenbasierten Ereignisquellen verarbeitet und in der servicespezifischen Dokumentation unter Lambda mit Ereignissen von anderen aufrufen AWS service.
- AWS Dienste — AWS Dienste können Ihre Funktion synchron oder asynchron aufrufen. Beim synchronen Aufruf entscheidet der Dienst, ob er es erneut versucht. Beispielsweise versucht Amazon S3 Batch Operations den Vorgang erneut, wenn die Lambda-Ereignisfunktion einen `TemporaryFailure` Antwortcode zurückgibt. Dienste, die Anfragen von einem Upstream-

Benutzer oder -Client weiterleiten, haben möglicherweise eine Wiederholungsstrategie oder leiten die Fehlerantwort an den Anforderer zurück. Zum Beispiel gibt API Gateway die Fehlerantwort immer an den Anforderer zurück.

Bei asynchronen Aufrufen ist die Wiederholungslogik identisch mit der Wiederholungslogik, unabhängig von der Aufrufquelle. Standardmäßig wiederholt Lambda einen fehlgeschlagenen asynchronen Aufruf bis zu zweimal. Weitere Informationen finden Sie unter Wie Lambda mit Fehlern und Wiederholungsversuchen mit asynchronem Aufruf umgeht.

- Andere Konten und Clients – Wenn Sie Zugriff auf andere Konten gewähren, können Sie mithilfe von ressourcenbasierten Richtlinien einschränken, welche Services und Ressourcen von ihnen zum Aufruf Ihrer Funktion konfiguriert werden können. Zum Schutz Ihrer Funktion vor einer Überlastung sollten Sie Ihrer Funktion mit Amazon API Gateway eine API-Ebene vorlagern. Hinweise zur Behandlung von API-Gateway-Fehlern finden Sie unterthe section called “Fehler”.

Um Ihnen bei der Behebung von Fehlern in Lambda-Anwendungen zu helfen, ist Lambda in Dienste wie Amazon CloudWatch und integriert. AWS X-Ray Sie können mithilfe einer Kombination aus Protokollen, Metriken, Alarmen und Ablaufverfolgung Probleme in Funktionscode, API und anderen Ressourcen, die für Ihre Anwendung erforderlich sind, schnell erkennen und aufdecken. Weitere Informationen finden Sie unter Überwachung, Debuggen und Problembehandlung von Lambda-Ereignisfunktionen.

Verwenden Sie die rekursive Lambda-Schleifenerkennung, um Endlosschleifen zu verhindern

Wenn Sie eine Lambda-Ereignisfunktion so konfigurieren, dass sie an denselben Dienst oder dieselbe Ressource ausgegeben wird, die die Funktion aufruft, ist es möglich, eine unendliche rekursive Schleife zu erstellen. Beispielsweise könnte eine Lambda-Ereignisfunktion eine Nachricht in eine Amazon Simple Queue Service (Amazon SQS) -Warteschlange schreiben, die dann dieselbe Funktion aufruft. Dieser Aufruf veranlasst die Funktion, eine weitere Nachricht in die Warteschlange zu schreiben, die wiederum erneut die Funktion aufruft.

Unbeabsichtigte rekursive Schleifen können dazu führen, dass Ihnen unerwartete Gebühren in Rechnung gestellt werden. AWS-Konto Außerdem können Schleifen dazu führen, dass Lambda skaliert wird und die gesamte verfügbare Parallelität Ihres Kontos nutzt. Um die Auswirkungen unbeabsichtigter Schleifen zu verringern, erkennt Lambda bestimmte Arten von rekursiven Schleifen kurz nach ihrem Auftreten.

Wenn Lambda eine rekursive Schleife erkennt, wird der Aufruf Ihrer Funktion standardmäßig gestoppt und Sie werden benachrichtigt. Wenn Ihr Entwurf bewusst rekursive Muster verwendet, können Sie die Standardkonfiguration einer Funktion so ändern, dass sie rekursiv aufgerufen werden kann. Weitere Informationen finden Sie unter the section called “Zulassen, dass eine Lambda-Ereignisfunktion in einer rekursiven Schleife ausgeführt wird”.

Sections

- Grundlegendes zur Erkennung rekursiver Schleifen
- Unterstützt AWS-Services und SDKs
- Benachrichtigungen zu rekursiven Schleifen
- Reagieren auf Benachrichtigungen im Zusammenhang mit der Erkennung rekursiver Schleifen
- Zulassen, dass eine Lambda-Ereignisfunktion in einer rekursiven Schleife ausgeführt wird
- Unterstützte Regionen für die Erkennung rekursiver Lambda-Schleifen

Grundlegendes zur Erkennung rekursiver Schleifen

Zur Erkennung rekursiver Schleifen in Lambda werden Ereignisse nachverfolgt. Lambda ist ein ereignisgesteuerter Compute-Service, der Ihren Funktionscode ausführt, wenn bestimmte Ereignisse auftreten. Ein Beispiel wäre etwa das Hinzufügen eines Elements zu einer Amazon-SQS-Warteschlange oder zu einem Amazon Simple Notification Service (Amazon SNS)-Thema.

Ereignisse werden von Lambda als JSON-Objekte mit Informationen zur Änderung des Systemstatus an Ihre Funktion übergeben. Die Ausführung Ihrer Funktion infolge eines Ereignisses wird als Aufruf bezeichnet.

Zur Erkennung rekursiver Schleifen verwendet Lambda AWS X-Ray-Ablaufverfolgungs-Header. Wenn AWS-Services , die die Erkennung rekursiver Schleifen unterstützen, Ereignisse an Lambda senden, werden diese Ereignisse automatisch mit Metadaten versehen. Wenn Ihre Lambda-Ereignisfunktion eines dieser Ereignisse in ein anderes schreibt, das AWS-Service mithilfe einer unterstützten Version eines AWS SDK unterstützt wird, aktualisiert sie diese Metadaten. In den aktualisierten Metadaten ist angegeben, wie oft das Ereignis die Funktion aufgerufen hat.

> **Note**
>
> Sie müssen die X-Ray aktive Ablaufverfolgung nicht aktivieren, damit diese Funktion funktioniert. Die Erkennung rekursiver Schleifen ist standardmäßig für alle AWS -Kunden aktiviert. Die Nutzung des Features ist kostenlos.

Bei einer Kette von Anforderungen handelt es sich um eine Sequenz von Lambda-Aufrufen, die durch das gleiche auslösende Ereignis verursacht werden. Stellen Sie sich zum Beispiel vor, dass eine Amazon SQS SQS-Warteschlange Ihre Lambda-Ereignisfunktion aufruft. Ihre Lambda-Ereignisfunktion sendet dann das verarbeitete Ereignis zurück an dieselbe Amazon SQS SQS-Warteschlange, die Ihre Funktion erneut aufruft. In diesem Beispiel gehört jeder Aufruf Ihrer Funktion zur gleichen Kette von Anforderungen.

Wenn Ihre Funktion ungefähr 16-mal in der gleichen Kette von Anforderungen aufgerufen wird, beendet Lambda automatisch den nächsten Funktionsaufruf in dieser Anforderungskette und benachrichtigt Sie. Wenn Ihre Funktion mit mehreren Auslösern konfiguriert ist, sind Aufrufe von anderen Auslösern nicht betroffen.

> **Note**
>
> Selbst wenn die `maxReceiveCount`-Einstellung in der Redrive-Richtlinie der Quellwarteschlange höher als 16 ist, verhindert der Lambda-Rekursionsschutz nicht, dass Amazon SQS die Nachricht erneut versucht, nachdem eine rekursive Schleife erkannt und beendet wurde. Wenn Lambda eine rekursive Schleife erkennt und nachfolgende Aufrufe abbricht, gibt es ein `RecursiveInvocationException` an die Zuordnung von Ereignisquellen zurück. Dadurch wird der `receiveCount`-Wert in der Nachricht

erhöht. Lambda versucht weiterhin, die Nachricht zu wiederholen, und blockiert weiterhin Funktionsaufrufe, bis Amazon SQS feststellt, dass das `maxReceiveCount` überschritten wurde, und die Nachricht an die konfigurierte Warteschlange für unzustellbare Nachrichten sendet.

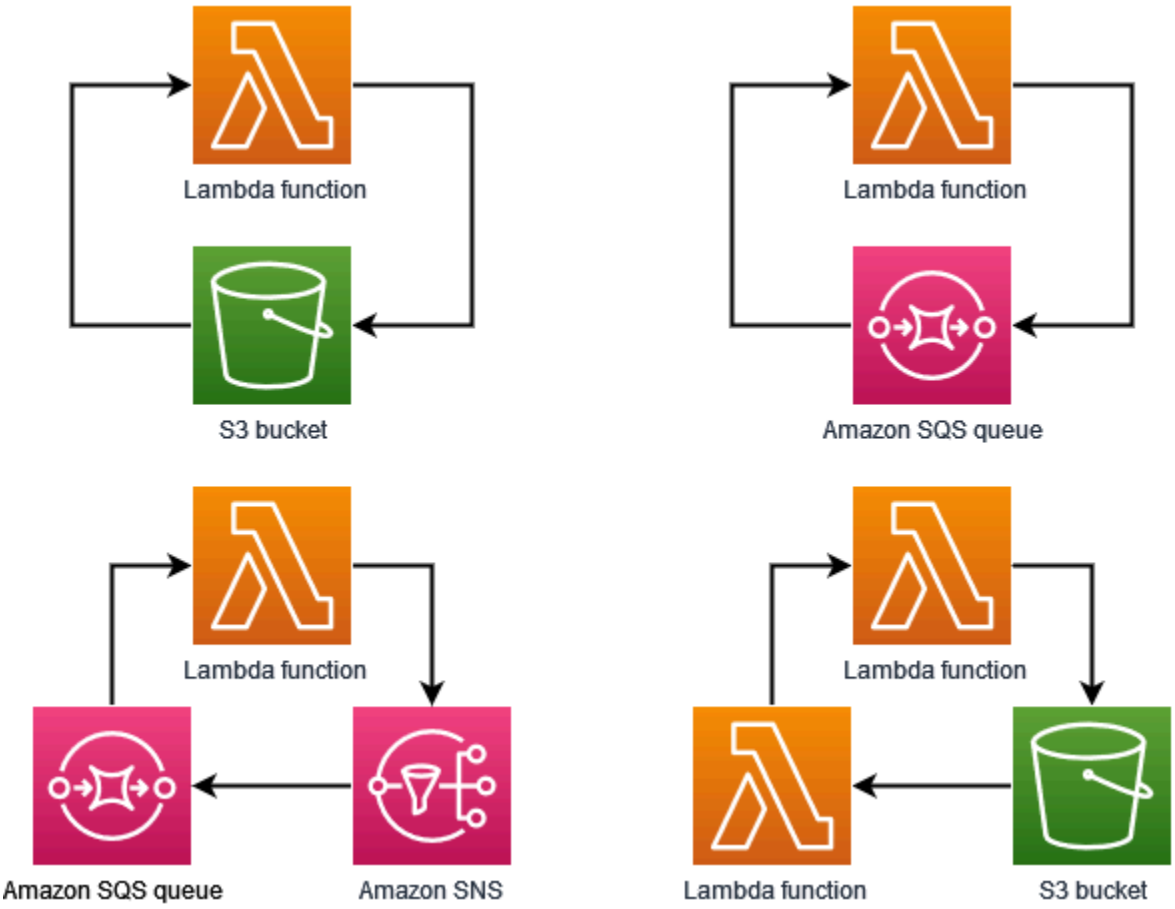
Wenn Sie für Ihre Event-Funktion ein Ziel bei einem Ausfall oder eine Warteschlange mit unbestätigten Briefen konfiguriert haben, sendet Lambda das Ereignis auch vom gestoppten Aufruf an Ihre Ziel- oder Dead-Letter-Warteschlange. Wenn Sie für Ihre Funktion ein Ziel oder eine Warteschlange für unzustellbare Nachrichten konfigurieren, achten Sie darauf, keinen Ereignisauslöser und keine Zuordnung von Ereignisquellen zu verwenden, die Ihre Funktion auch verwendet. Wenn Sie Ereignisse an dieselbe Ressource senden, die Ihre Funktion aufruft, können Sie eine weitere rekursive Schleife erstellen, und diese Schleife wird ebenfalls beendet. Wenn Sie die Erkennung von Rekursionsschleifen deaktivieren, wird diese Schleife nicht beendet.

Unterstützt AWS-Services und SDKs

Lambda kann nur rekursive Schleifen erkennen, die bestimmte unterstützte enthalten. AWS-Services Damit rekursive Schleifen erkannt werden können, muss Ihre Event-Funktion auch eines der unterstützten SDKs verwenden. AWS

Unterstützt AWS-Services

Lambda erkennt derzeit rekursive Schleifen zwischen Ihren Funktionen, Amazon SQS, Amazon S3, Amazon SNS und Amazon EventBridge Custom Event Buses. Lambda erkennt auch Schleifen, die nur aus Lambda-Ereignisfunktionen bestehen, die sich gegenseitig synchron oder asynchron aufrufen können. Die folgenden Diagramme zeigen einige Beispiele für Schleifen, die von Lambda erkannt werden:



Wenn ein anderes AWS-Service System wie Amazon DynamoDB Teil der Schleife ist, kann Lambda es derzeit nicht erkennen und stoppen.

Da Lambda derzeit nur rekursive Schleifen erkennt, an denen Amazon SQS-, Amazon S3-, Amazon SNS- und Amazon EventBridge Custom-Ereignisbusse beteiligt sind, ist es immer noch möglich, dass Schleifen, an denen andere beteiligt sind, zu einer unbeabsichtigten Nutzung Ihrer Lambda-Ereignisfunktion führen AWS-Services können.

Um zu verhindern, dass Ihnen unerwartete Gebühren in Rechnung gestellt werden AWS-Konto, empfehlen wir Ihnen, CloudWatchAmazon-Alarme so zu konfigurieren, dass Sie auf ungewöhnliche Nutzungsmuster aufmerksam gemacht werden. Sie können beispielsweise so konfigurieren, dass Sie über CloudWatch Spitzenwerte bei der Parallelität oder bei Aufrufen von Lambda Event Functions benachrichtigt werden. Des Weiteren können Sie einen Abrechnungsalarm konfigurieren, um benachrichtigt zu werden, wenn die Ausgaben für Ihr Konto einen von Ihnen angegebenen Schwellenwert übersteigen. Eine weitere Möglichkeit ist die Verwendung der AWS Cost Anomaly Detection, um über ungewöhnliche Abrechnungsmuster informiert zu werden.

Unterstützt AWS SDKs

Damit Lambda rekursive Schleifen erkennt, muss Ihre Funktion eine der folgenden SDK-Mindestversionen verwenden:

Laufzeit	AWS Erforderliche Mindestversion des SDK
Node.js	2.1147.0 (SDK-Version 2) 3.105.0 (SDK-Version 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 und Java 11	2.17.135
Java 17	2,20,81
Java 21	2,21,24
.NET	3,7,293,0
Ruby	3,134,0
PHP	3,232,0
Go	V2 SDK 1.57.0

Einige Lambda-Laufzeiten wie Python Node.js enthalten eine Version des AWS SDK. Wenn die in der Laufzeit Ihrer Funktion enthaltene SDK-Version kleiner ist als die erforderliche Mindestversion, können Sie dem Bereitstellungspaket Ihrer Funktion eine unterstützte Version des SDK hinzufügen. Sie können auch eine Lambda-Ebene verwenden, um Ihrer Funktion eine unterstützte SDK-Version hinzuzufügen. Eine Liste der SDKs, die in der jeweiligen Lambda-Laufzeitumgebung enthalten sind, finden Sie unter Laufzeiten der Lambda-Event-Funktion.

Benachrichtigungen zu rekursiven Schleifen

Wenn Lambda eine rekursive Schleife beendet, erhalten Sie Benachrichtigungen über das Health Dashboard und per E-Mail. Sie können auch CloudWatch Metriken verwenden, um die Anzahl der rekursiven Aufrufe zu überwachen, die Lambda gestoppt hat.

Health Dashboard Benachrichtigungen

Wenn Lambda einen rekursiven Aufruf stoppt, wird auf der Seite „Ihr Kontostatus“ unter „Offene Probleme“ und „Aktuelle Probleme“ eine Benachrichtigung Health Dashboard angezeigt. Beachten Sie, dass es nach der Beendigung eines rekursiven Aufrufs durch Lambda bis zu 3,5 Stunden dauern kann, bis diese Benachrichtigung angezeigt wird. Weitere Informationen zum Anzeigen von Kontoereignissen in der Health Dashboard finden Sie unter Erste Schritte mit Ihrem AWS Health Dashboard — Ihr Kontostatus im AWS Health-Benutzerhandbuch.

E-Mail-Benachrichtigungen

Wenn Lambda einen rekursiven Aufruf Ihrer Event-Funktion zum ersten Mal stoppt, sendet es Ihnen eine E-Mail-Warnung. Für jede Funktion in Ihrem AWS-Konto wird alle 24 Stunden maximal eine E-Mail gesendet. Nachdem Lambda eine E-Mail-Benachrichtigung gesendet hat, erhalten Sie für einen Zeitraum von 24 Stunden keine E-Mails mehr zu dieser Funktion. Das gilt auch, wenn Lambda weitere rekursive Aufrufe der Funktion beendet. Beachten Sie, dass es nach der Beendigung eines rekursiven Aufrufs durch Lambda bis zu 3,5 Stunden dauern kann, bis Sie diese E-Mail-Benachrichtigung erhalten.

Lambda sendet rekursive Loop-E-Mail-Benachrichtigungen an Ihren primären AWS-Konto Kundenkontakt und an den alternativen Betriebskontakt. Informationen zum Anzeigen oder Aktualisieren der E-Mail-Adressen in Ihrem Konto finden Sie in der allgemeinen AWS -Referenz unter Updating contact information.

CloudWatch Amazon-Metriken

Die CloudWatch Metrik `RecursiveInvocationsDropped` zeichnet die Anzahl der Funktionsaufrufen auf, die Lambda gestoppt hat, weil Ihre Funktion in einer einzigen Anforderungskette mehr als etwa 16 Mal aufgerufen wurde. Lambda gibt diese Metrik aus, sobald ein rekursiver Aufruf beendet wurde. Um diese Metrik anzuzeigen, folgen Sie den Anweisungen unter Metriken anzeigen auf der CloudWatch Konsole und wählen Sie die Metrik aus. `RecursiveInvocationsDropped`

Reagieren auf Benachrichtigungen im Zusammenhang mit der Erkennung rekursiver Schleifen

Wenn Ihre Event-Funktion mehr als ungefähr 16 Mal durch dasselbe auslösende Ereignis aufgerufen wird, stoppt Lambda den nächsten Funktionsaufruf für dieses Ereignis, um die rekursive Schleife zu unterbrechen. Gehen Sie wie folgt vor, um zu verhindern, dass eine rekursive Schleife, die von Lambda unterbrochen wurde, erneut auftritt:

- Reduzieren Sie die verfügbare Parallelität Ihrer Funktion auf Null. Dadurch werden alle zukünftigen Aufrufe gedrosselt.
- Entfernen oder deaktivieren Sie den Auslöser oder die Zuordnung von Ereignisquellen, der bzw. die Ihre Funktion aufruft.
- Identifizieren und beheben Sie Codefehler, die Ereignisse in die AWS Ressource zurückschreiben, die Ihre Funktion aufruft. Eine Fehlerquelle entsteht häufig, wenn die Ereignisquelle und das Ziel einer Funktion mithilfe von Variablen definiert werden. Achten Sie darauf, nicht für beide Variablen den gleichen Wert zu verwenden.

Wenn es sich bei der Ereignisquelle für Ihre Lambda-Ereignisfunktion um eine Amazon SQS SQS-Warteschlange handelt, sollten Sie außerdem erwägen, eine Warteschlange für unzustellbare Briefe in der Quellwarteschlange zu konfigurieren.

> **Note**
>
> Stellen Sie sicher, dass Sie die Warteschlange für unzustellbare Briefe in der Quellwarteschlange konfigurieren, nicht in der Lambda-Ereignisfunktion. Die Warteschlange für unzustellbare Nachrichten, die Sie für eine Funktion konfigurieren, wird für die Warteschlange asynchroner Aufrufe der Funktion und nicht für Ereignisquellen-Warteschlangen verwendet.

Wenn es sich bei der Ereignisquelle um ein Amazon-SNS-Thema handelt, empfiehlt es sich gegebenenfalls, für Ihre Funktion ein Ziel bei Ausfall hinzuzufügen.

Um die verfügbare Parallelität Ihrer Event-Funktion auf Null zu reduzieren (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion.

3. Wählen Sie Drosseln aus.
4. Wählen Sie im Dialogfeld Funktion drosseln die Option Bestätigen aus.

Um eine Trigger- oder Ereignisquellenzuordnung für Ihre Event-Funktion (Konsole) zu entfernen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion.
3. Wählen Sie die Registerkarte Konfiguration und anschließend Auslöser aus.
4. Wählen Sie unter Auslöser den Auslöser oder die Zuordnung von Ereignisquellen aus, den bzw. die Sie löschen möchten, und wählen Sie anschließend Löschen aus.
5. Wählen Sie im Dialogfeld Auslöser löschen die Option Löschen aus.

Um die Zuordnung einer Ereignisquelle für Ihre Event-Funktion zu deaktivieren (AWS CLI)

1. Um die UUID für die Ereignisquellenzuordnung zu finden, die Sie deaktivieren möchten, führen Sie den Befehl AWS Command Line Interface (AWS CLI) list-event-source-mappings aus.

   ```
   aws lambda list-event-source-mappings
   ```

2. Um die Zuordnung der Ereignisquellen zu deaktivieren, führen Sie den folgenden Befehl update-event-source-mapping aus. AWS CLI

   ```
   aws lambda update-event-source-mapping --function-name MyFunction \
   --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
   ```

Zulassen, dass eine Lambda-Ereignisfunktion in einer rekursiven Schleife ausgeführt wird

Wenn Ihr Design absichtlich eine rekursive Schleife verwendet, können Sie eine Lambda-Ereignisfunktion so konfigurieren, dass sie rekursiv aufgerufen werden kann. Wir empfehlen, dass Sie rekursive Schleifen in Ihrem Entwurf vermeiden. Implementierungsfehler können dazu führen, dass rekursive Aufrufe Ihre gesamte verfügbare Parallelität nutzen und AWS-Konto dass Ihrem Konto unerwartete Gebühren in Rechnung gestellt werden.

> **Important**
>
> Wenn Sie rekursive Schleifen verwenden, sollten Sie diese mit Vorsicht behandeln. Implementieren Sie bewährte Leitplanken, um das Risiko von Implementierungsfehlern zu minimieren. Weitere Informationen zu bewährten Methoden für die Verwendung rekursiver Muster finden Sie bei Serverless Land unter Recursive patterns that cause run-away Lambda functions.

Mithilfe der Lambda-Konsole, der AWS Command Line Interface (AWS CLI) und der API können Sie Funktionen so konfigurieren, dass sie rekursive Schleifen zulassen. PutFunctionRecursionConfig Sie können auch die Einstellung für die Erkennung rekursiver Schleifen einer Funktion in und konfigurieren. AWS SAM CloudFormation

Standardmäßig erkennt und beendet Lambda rekursive Schleifen. Sofern Ihr Entwurf nicht bewusst eine rekursive Schleife verwendet, empfehlen wir Ihnen, die Standardkonfiguration Ihrer Funktionen nicht zu ändern.

Beachten Sie, dass die CloudWatch Metrik `RecursiveInvocationsDropped` nicht ausgegeben wird, wenn Sie eine Funktion so konfigurieren, dass sie rekursive Schleifen zulässt.

Console

Um zu ermöglichen, dass eine Ereignisfunktion in einer rekursiven Schleife ausgeführt wird (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion, um die Funktionsdetailseite zu öffnen.
3. Wählen Sie die Registerkarte Konfiguration und anschließend Gleichzeitigkeit und Rekursionserkennung.
4. Wählen Sie neben Rekursive Schleifenerkennung die Option Bearbeiten aus.
5. Wählen Sie Rekursive Schleifen zulassen aus.
6. Wählen Sie Speichern.

AWS CLI

Sie können die PutFunctionRecursionConfigAPI verwenden, um zu ermöglichen, dass Ihre Funktion in einer rekursiven Schleife aufgerufen wird. Geben Sie `Allow` als Parameter für die

rekursive Schleife an. Sie können diese API beispielsweise mit dem folgenden Befehl aufrufen: `put-function-recursion-config` AWS CLI

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Allow
```

Sie können die Konfiguration Ihrer Event-Funktion auf die Standardeinstellung zurücksetzen, sodass Lambda rekursive Schleifen beendet, wenn es sie erkennt. Bearbeiten Sie die Konfiguration Ihrer Funktion mit der Lambda-Konsole oder dem AWS CLI.

Console

Um eine Event-Funktion so zu konfigurieren, dass rekursive Schleifen beendet werden (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion, um die Funktionsdetailseite zu öffnen.
3. Wählen Sie die Registerkarte Konfiguration und anschließend Gleichzeitigkeit und Rekursionserkennung.
4. Wählen Sie neben Rekursive Schleifenerkennung die Option Bearbeiten aus.
5. Wählen Sie Rekursive Schleifen beenden aus.
6. Wählen Sie Speichern.

AWS CLI

Sie können die PutFunctionRecursionConfigAPI verwenden, um Ihre Funktion so zu konfigurieren, dass Lambda rekursive Schleifen beendet, wenn es sie erkennt. Geben Sie `Terminate` als Parameter für die rekursive Schleife an. Sie können diese API beispielsweise mit dem folgenden Befehl aufrufen: `put-function-recursion-config` AWS CLI

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Terminate
```

Unterstützte Regionen für die Erkennung rekursiver Lambda-Schleifen

Die rekursive Lambda-Schleifenerkennung wird in allen kommerziellen Regionen unterstützt.

Erstellen und Verwalten von Lambda-Funktions-URLs

Eine Funktions-URL ist ein dedizierter HTTP (S) -Endpunkt für Ihre Lambda-Ereignisfunktion. Sie können eine Funktions-URL über die Lambda-Konsole oder die Lambda-API erstellen und konfigurieren.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: Funktions-URLs und Amazon API Gateway. Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “Funktions-URLs im Vergleich zu Amazon API Gateway”.

Wenn Sie eine Funktions-URL erstellen, generiert Lambda automatisch einen eindeutigen URL-Endpunkt für Sie. Sobald Sie eine Funktions-URL erstellt haben, ändert sich ihr URL-Endpunkt nie mehr. Funktions-URL-Endpunkte haben das folgende Format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Funktions-URLs werden nicht in allen AWS Regionen unterstützt. Informationen zur Verfügbarkeit finden Sie unter AWS Regionaldienste und filtern Sie nach`Feature Name = Function URLs`.

Funktions-URLs sind Dual-Stack-fähig und unterstützen IPv4 und IPv6. Nachdem Sie eine Funktions-URL für Ihre Funktion konfiguriert haben, können Sie Ihre Funktion über ihren HTTP (S) -Endpunkt aufrufen, indem Sie einen Webbrowser, Curl, Postman oder einen beliebigen HTTP-Client verwenden.

> **Note**
>
> Auf Ihre Funktions-URL können Sie nur über das öffentliche Internet zugreifen. Lambda-Ereignisfunktionen unterstützen zwar AWS PrivateLink, Funktions-URLs jedoch nicht.

URLs der Lambda-Funktion verwenden ressourcenbasierte Richtlinien für Sicherheit und Zugriffskontrolle. Funktions-URLs unterstützen auch CORS-Konfigurationsoptionen (Cross-Origin Resource Sharing).

Sie können Funktions-URLs auf jeden Funktionsalias oder auf die `$LATEST` unveröffentlichte Funktionsversion anwenden. Sie können keiner anderen Funktionsversion eine Funktions-URL hinzufügen.

Im folgenden Abschnitt wird gezeigt, wie Sie eine Funktions-URL mithilfe der Lambda-Konsole AWS CLI, und der CloudFormation Vorlage erstellen und verwalten. Eine schrittweise Anleitung zur Verwendung von Funktions-URLs als Webhook-Endpunkte finden Sie unter. the section called “Tutorial: Erstellen eines Webhook-Endpunkts”

Themen

- Erstellen einer Funktions-URL (Konsole)
- Eine Funktions-URL erstellen (AWS CLI)
- Hinzufügen einer Funktions-URL zu einer CloudFormation Vorlage
- Cross-origin Gemeinsame Nutzung von Ressourcen (CORS)
- Drosseln von Funktions-URLs
- Deaktivieren von Funktions-URLs
- Löschen von Funktions-URLs
- Kontrolle des Zugriffs auf Lambda-Funktions-URLs
- Aufrufen von Lambda-Funktions-URLs
- Überwachen der Lambda-Funktions-URLs
- Wählen Sie eine Methode aus, um Ihre Lambda-Ereignisfunktion mithilfe einer HTTP-Anfrage aufzurufen
- Tutorial: Erstellen eines Webhook-Endpunkts mithilfe einer Lambda-Funktions-URL

Erstellen einer Funktions-URL (Konsole)

Gehen Sie wie folgt vor, um eine Funktions-URL mit der Konsole zu erstellen.

So erstellen Sie eine Funktions-URL für eine vorhandene Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Wählen Sie den Namen der Event-Funktion, für die Sie die Funktions-URL erstellen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
4. Wählen Sie Funktions-URL erstellen.
5. Für Auth-Typ wählen Sie AWS_IAM oder KEINE aus. Weitere Informationen zur Authentifizierung der Funktions-URL finden Sie unter Zugriffskontrolle.
6. (Optional) Wählen Sie Configure cross-origin resource sharing (CORS) (Cross-Origin Resource Sharing (CORS) konfigurieren) aus und konfigurieren Sie dann die CORS-Einstellungen für Ihre Funktions-URL. Weitere Informationen über CORS finden Sie unter Cross-origin Gemeinsame Nutzung von Ressourcen (CORS).
7. Wählen Sie Speichern.

Dies erstellt eine Funktions-URL für die `$LATEST` unveröffentlichte Version Ihrer Funktion. Die Funktions-URL wird im Abschnitt Function overview (Funktionsübersicht) der Konsole angezeigt.

So erstellen Sie eine Funktions-URL für einen vorhandenen Alias

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Event-Funktion mit dem Alias, für den Sie die Funktions-URL erstellen möchten.
3. Wählen Sie die Registerkarte Aliase und wählen Sie dann den Namen der Funktion mit dem Alias, für die Sie die Funktions-URL erstellen möchten.
4. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
5. Wählen Sie Funktions-URL erstellen.
6. Für Auth-Typ wählen Sie AWS_IAM oder KEINE aus. Weitere Informationen zur Authentifizierung der Funktions-URL finden Sie unter Zugriffskontrolle.
7. (Optional) Wählen Sie Configure cross-origin resource sharing (CORS) (Cross-Origin Resource Sharing (CORS) konfigurieren) aus und konfigurieren Sie dann die CORS-Einstellungen für Ihre Funktions-URL. Weitere Informationen über CORS finden Sie unter Cross-origin Gemeinsame Nutzung von Ressourcen (CORS).
8. Wählen Sie Speichern.

Dies erstellt eine Funktions-URL für Ihren Funktionsalias. Die Funktions-URL wird im Abschnitt Funktionsübersicht der Konsole für Ihren Alias angezeigt.

So erstellen Sie eine neue Funktion mit einem Funktions-URL

Um eine neue Event-Funktion mit einer Funktions-URL (Konsole) zu erstellen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie unter Funktionsname einen Namen für Ihre Funktion ein, z. B. **`my-function`**.
   b. Wählen Sie für Runtime die Sprache Runtime aus, die Sie bevorzugen, z. B. Node.js 24.
   c. Wählen Sie für Architecture (Architektur) entweder x86_64 oder arm64 aus.
   d. Erweitern Sie Permissions (Berechtigungen) und wählen Sie dann aus, ob eine neue Ausführungsrolle erstellt oder eine vorhandene Rolle verwendet werden soll.
4. Erweitern Sie Advanced settings (Erweiterte Einstellungen) und wählen Sie dann Funktions-URL aus.
5. Wählen Sie als Auth type (Authentifizierungstyp) die Option AWS_IAM (AWS_IAM) oder NONE (KEINE) aus. Weitere Informationen zur Authentifizierung der Funktions-URL finden Sie unter Zugriffskontrolle.
6. (Optional) Wählen Sie Cross-Origin Resource Sharing (CORS) konfigurieren aus. Wenn Sie diese Option während der Funktionserstellung auswählen, lässt Ihre Funktions-URL standardmäßig sämtliche Anfragen zu. Sie können die CORS-Einstellungen für Ihre Funktions-URL bearbeiten, nachdem Sie die Funktion erstellt haben. Weitere Informationen über CORS finden Sie unter Cross-origin Gemeinsame Nutzung von Ressourcen (CORS).
7. Wählen Sie „Event-Funktion erstellen“.

Dies erstellt eine neue Funktions mit einer Funktions-URL für die `$LATEST` unveröffentlichte Version Ihrer Funktion. Die Funktions-URL wird im Abschnitt Function overview (Funktionsübersicht) der Konsole angezeigt.

Eine Funktions-URL erstellen (AWS CLI)

Führen Sie den folgenden Befehl aus, um eine Funktions-URL für eine bestehende Lambda-Ereignisfunktion mithilfe von AWS Command Line Interface (AWS CLI) zu erstellen:

```
aws lambda create-function-url-config \
    --function-name my-function \
```

```
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

Dies fügt eine Funktions-URL zum **`prod`**-Qualifizierer für die Funktion**`my-function`** hinzu. Weitere Informationen zu diesen Konfigurationsparametern finden Sie unter CreateFunctionUrlConfig in der API-Referenz.

> **Note**
>
> Um eine Funktions-URL mithilfe von zu erstellen AWS CLI, muss die Funktion bereits vorhanden sein.

Hinzufügen einer Funktions-URL zu einer CloudFormation Vorlage

Verwenden Sie die folgende Syntax, um Ihrer CloudFormation Vorlage eine `AWS::Lambda::Url` Ressource hinzuzufügen:

JSON

```
{
  "Type" : "AWS::Lambda::Url",
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
    Cors
  Qualifier: String
  TargetFunctionArn: String
```

Parameters

- (Erforderlich) `AuthType` – Definiert den Typ der Authentifizierung für Ihre Funktions-URL. Die möglichen Werte sind `AWS_IAM` oder `NONE`. Um den Zugriff nur auf authentifizierte Benutzer zu beschränken, setzen Sie den Wert auf `AWS_IAM`. Um die IAM-Authentifizierung zu umgehen und jedem Benutzer zu erlauben, Anfragen an Ihre Funktion zu stellen, setzten Sie dies auf `NONE`.
- (Optional)`Cors` – Definiert die CORS-Einstellungen für Ihre Funktions-URL. Verwenden Sie `Cors` die folgende Syntax CloudFormation, um Ihrer `AWS::Lambda::Url` Ressource in etwas hinzuzufügen.

  Example AWS::Lambda:: Url.Cors (JSON)

  ```
  {
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
    "AllowMethods" : [ String, ... ],
    "AllowOrigins" : [ String, ... ],
    "ExposeHeaders" : [ String, ... ],
    "MaxAge" : Integer
  }
  ```

  Example AWS::Lambda:: Url.Cors (YAML)

  ```
    AllowCredentials: Boolean
    AllowHeaders:
      - String
    AllowMethods:
      - String
    AllowOrigins:
      - String
    ExposeHeaders:
      - String
    MaxAge: Integer
  ```

- (Optional) `Qualifier` – Der Aliasname.
- (Erforderlich) `TargetFunctionArn` — Der Name oder der Amazon-Ressourcenname (ARN) der Lambda-Ereignisfunktion. Gültige Namensformate sind unter anderem:
  - Funktionsname – `my-function`.
  - Funktions-ARN – `arn:aws:lambda:us-west-2:123456789012:function:my-function`.

- Partiellet ARN – `123456789012:function:my-function`.

Cross-origin Gemeinsame Nutzung von Ressourcen (CORS)

Um zu definieren, wie verschiedene Ursprünge auf Ihre Funktions-URL zugreifen können, verwenden Sie Cross-Origin Resource Sharing (CORS). Wir empfehlen, CORS zu konfigurieren, wenn Sie beabsichtigen, Ihre Funktions-URL von einer anderen Domain aus aufzurufen. Lambda unterstützt die folgenden CORS-Header für Funktions-URLs.

CORS-Header	CORS-Konfigurationseigenschaft	Beispielwerte

Access-Control-Allow-Origin	`AllowOrigins`	`*` (alle Ursprünge zulassen)
		`https://www.example.com`
		`http://localhost:60905`

Access-Control-Allow-Methods	`AllowMethods`	`GET`, `POST`, `DELETE`, `*`
Access-Control-Allow-Headers	`AllowHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Expose-Headers	`ExposeHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Allow-Credentials	`AllowCredentials`	`TRUE`
Access-Control-Max-Age	`MaxAge`	`5` (Standard), `300`

Wenn Sie CORS für eine Funktions-URL mithilfe der Lambda-Konsole oder der konfigurieren AWS CLI, fügt Lambda über die Funktions-URL automatisch die CORS-Header zu allen Antworten hinzu. Alternativ können Sie Ihrer Funktionsantwort manuell CORS-Header hinzufügen. Wenn es widersprüchliche Header gibt, ist das erwartete Verhalten abhängig vom Typ der Anfrage:

- Bei Preflight-Anfragen wie OPTIONS-Anfragen haben die konfigurierten CORS-Header auf der Funktions-URL Vorrang. Lambda gibt nur diese CORS-Header in der Antwort zurück.
- Bei Nicht-Preflight-Anforderungen wie GET- oder POST-Anforderungen gibt Lambda sowohl die konfigurierten CORS-Header der Funktions-URL als auch die von der Funktion zurückgegebenen CORS-Header zurück. Dies kann zu doppelten CORS-Header in der Antwort führen. Möglicherweise wird ein Fehler ähnlich dem folgenden angezeigt: `The 'Access-Control-Allow-Origin' header contains multiple values '*, *', but only one is allowed.`

Im Allgemeinen empfehlen wir, alle CORS-Einstellungen auf der Funktions-URL zu konfigurieren, anstatt CORS-Header manuell in der Funktionsantwort zu senden.

Drosseln von Funktions-URLs

Die Drosselungsrate begrenzt die Geschwindigkeit, mit der Ihre Funktion Anfragen verarbeitet. Dies ist in vielen Situationen nützlich, z. B. um zu verhindern, dass Ihre Funktion nachgelagerte Ressourcen überlastet, oder beim Verarbeiten eines plötzlichen Anstiegs der Anfragen.

Sie können die Anzahl der Anfragen, die Ihre Lambda Event Function über eine Funktions-URL verarbeitet, drosseln, indem Sie reservierte Parallelität konfigurieren. Reservierte Gleichzeitigkeit begrenzt die Anzahl der maximalen gleichzeitigen Aufrufe für Ihre Funktion. Die maximale Anforderungsrate Ihrer Funktion pro Sekunde (RPS) entspricht dem Zehnfachen der konfigurierten reservierten Gleichzeitigkeit. Wenn Sie Ihre Funktion beispielsweise mit einer reservierten Gleichzeitigkeit von 100 konfigurieren, beträgt der maximale RPS 1 000.

Immer wenn Ihre Funktionsparallelität die reservierte Gleichzeitigkeit überschreitet, gibt Ihre Funktions-URL einen HTTP-429-Statuscode zurück. Wenn Ihre Funktion eine Anforderung erhält, die das 10-fache RPS-Maximum basierend auf Ihrer konfigurierten reservierten Gleichzeitigkeit überschreitet, erhalten Sie auch einen HTTP-429-Fehler. Weitere Hinweise zur reservierten Gleichzeitigkeit finden Sie unter Konfiguration reservierter Parallelität für eine Event-Funktion.

Deaktivieren von Funktions-URLs

Im Notfall sollten Sie den gesamten Datenverkehr zu Ihrer Funktions-URL ablehnen. Um eine Funktion zu deaktivieren, setzen Sie die reservierte Gleichzeitigkeit auf Null. Dies drosselt alle Anfragen an Ihre Funktions-URL, was zu HTTP-429-Statusantworten führt. Um Ihre Funktions-URL zu reaktivieren, löschen Sie die Konfiguration für die reservierte Gleichzeitigkeit oder legen Sie die Konfiguration auf einen Betrag größer als null fest.

Löschen von Funktions-URLs

Wenn Sie eine Funktions-URL löschen, können Sie sie nicht wiederherstellen. Das Erstellen einer neuen Funktions-URL führt zu einer anderen URL-Adresse.

> **Note**
>
> Wenn Sie Ihre Funktions-URL mit Authentifizierungstyp `NONE` löschen, löscht Lambda nicht automatisch die zugehörige ressourcenbasierte Richtlinie. Wenn Sie diese Richtlinie löschen möchten, müssen Sie dies manuell tun.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Event-Funktion.
3. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
4. Wählen Sie Löschen aus.
5. Geben Sie den Ausdruck delete (löschen) in das Feld ein, um den Löschvorgang zu bestätigen.
6. Wählen Sie Löschen aus.

> **Note**
>
> Wenn Sie eine Funktion löschen, die eine Funktions-URL hat, löscht Lambda asynchron die Funktions-URL. Wenn Sie sofort eine neue Funktion mit demselben Namen in demselben Konto erstellen, ist es möglich, dass die ursprüngliche Funktions-URL der neuen Funktion zugeordnet und nicht gelöscht wird.

Kontrolle des Zugriffs auf Lambda-Funktions-URLs

> **Note**
>
> Ab Oktober 2025 benötigen neue Funktions-URLs sowohl die Berechtigung `lambda:InvokeFunctionUrl` als auch `lambda:InvokeFunction`.

Sie können den Zugriff auf Ihre Lambda-Funktions-URLs mithilfe des AuthTypeParameters in Kombination mit ressourcenbasierten Richtlinien steuern, die an Ihre spezifische Event-Funktion

angehängt sind. Die Konfiguration dieser beiden Komponenten bestimmt, wer andere administrative Aktionen für Ihre Funktions-URL aufrufen oder ausführen kann.

Der `AuthType`-Parameter bestimmt, wie Lambda Anfragen an Ihre Funktions-URL authentifiziert oder autorisiert. Wenn Sie Ihre Funktions-URL konfigurieren, müssen Sie eine der folgenden `AuthType`-Optionen angeben:

- `AWS_IAM`— Lambda verwendet AWS Identity and Access Management (IAM), um Anfragen auf der Grundlage der Identitätsrichtlinie des IAM-Prinzipals und der ressourcenbasierten Richtlinie der Funktion zu authentifizieren und zu autorisieren. Wählen Sie diese Option, wenn Sie möchten, dass nur authentifizierte -Benutzer und -Rollen Ihre Funktion über die Funktions-URL aufrufen können.

- `NONE` – Lambda führt keine Authentifizierung durch, bevor Sie Ihre Funktion aufrufen. Die ressourcenbasierte Richtlinie Ihrer Funktion ist jedoch immer in Kraft und muss öffentlichen Zugriff gewähren, bevor Ihre Funktions-URL Anfragen erhalten kann. Wählen Sie diese Option, um öffentlichen, nicht authentifizierten Zugriff auf Ihre Funktions-URL zu ermöglichen.

Für zusätzliche Einblicke in die Sicherheit können Sie eine umfassende Analyse des externen AWS Identity and Access Management Access Analyzer Zugriffs auf Ihre Funktions-URL nutzen. IAM Access Analyzer überwacht auch, ob neue oder aktualisierte Berechtigungen für Ihre Lambda-Ereignisfunktionen vorliegen, um Sie bei der Identifizierung von Berechtigungen zu unterstützen, die öffentlichen und kontoübergreifenden Zugriff gewähren. Sie können IAM Access Analyzer kostenlos nutzen. Weitere Informationen zu den ersten Schritten mit IAM Access Analyzer finden Sie unter Verwenden des AWS IAM Access Analyzer.

Diese Seite enthält Beispiele für ressourcenbasierte Richtlinien für beide Authentifizierungstypen und wie diese Richtlinien mithilfe des AddPermissionAPI-Vorgangs oder der Lambda-Konsole erstellt werden. Informationen darüber, wie Sie Ihre Funktions-URL aufrufen können, nachdem Sie Berechtigungen eingerichtet haben, finden Sie unter Aufrufen von Lambda-Funktions-URLs.

Themen

- Verwenden des Authentifizierungstyps AWS_IAM
- Verwenden Sie den Authentifizierungstyp NONE
- Governance und Zugriffskontrolle

Verwenden des Authentifizierungstyps AWS_IAM

Wenn Sie den Authentifizierungstyp `AWS_IAM` auswählen, müssen Benutzer, die Ihre Lambda-Funktions-URL aufrufen, die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` haben. Je nachdem, wer die Aufrufanforderung stellt, müssen Sie diese Berechtigung möglicherweise mithilfe einer ressourcenbasierten Richtlinie erteilen.

Wenn sich der Principal, der die Anfrage stellt, in derselben URL befindet AWS-Konto wie die Funktions-URL, muss der Prinzipal entweder über `lambda:InvokeFunctionUrl` `lambda:InvokeFunction` Berechtigungen in seiner identitätsbasierten Richtlinie verfügen oder ihm in der ressourcenbasierten Richtlinie der Funktion Berechtigungen gewährt werden. Mit anderen Worten, eine ressourcenbasierte Richtlinie ist optional, wenn der Benutzer bereits die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` in seiner identitätsbasierten Richtlinie hat. Die Richtlinienbewertung folgt den Regeln, die in der Auswertungslogik für Richtlinien beschrieben sind.

Wenn der Prinzipal, der die Anfrage stellt, in einem anderen Konto ist, muss der Prinzipal sowohl eine identitätsbasierte Richtlinie haben, die ihm die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` gewährt, als auch Berechtigungen, die ihm in einer ressourcenbasierten Richtlinie für die Funktion gewährt werden, die er aufrufen möchte. Die Richtlinienbewertung folgt den im Abschnitt Festlegen, ob eine kontoübergreifende Anforderung zulässig ist erläuterten Regeln.

Die folgende ressourcenbasierte Richtlinie ermöglicht es der `example` Rolle in, die der Funktion AWS-Konto `444455556666` zugeordnete Funktions-URL aufzurufen. `my-function` Der Schlüssel lambda: InvokedViaFunctionUrl context beschränkt die `lambda:InvokeFunction` Aktion auf URL-Aufrufe von Funktionen. Das bedeutet, dass der Prinzipal die Funktions-URL verwenden muss, um die Funktion aufzurufen. Wenn Sie `lambda:InvokedViaFunctionUrl` nicht einfügen, kann der Prinzipal Ihre Funktion zusätzlich zur Funktions-URL auch über andere Aufrufmethoden aufrufen.

Example— Cross-account ressourcenbasierte Richtlinie

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
```

```
      },
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "AWS_IAM"
        }
      }
    },
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

Sie können diese ressourcenbasierte Richtlinie über die Konsole erstellen, indem Sie die folgenden Schritte ausführen:

So erteilen Sie einem anderen Konto (Konsole) URL-Aufrufberechtigungen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Funktion aus, für die Sie URL-Aufrufberechtigungen gewähren möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Resource-based Richtlinie die Option Berechtigungen hinzufügen aus.
5. Wählen Sie Funktions-URL aus.
6. Wählen Sie für Auth type (Authentifizierungstyp) die Option AWS_IAM aus.
7. Geben Sie eine Anweisungs-ID für Ihre Richtlinienanweisung ein.

8. Geben Sie unter Prinzipal den Amazon-Ressourcennamen (ARN) des Benutzers oder der Rolle ein, dem bzw. der Sie Berechtigungen gewähren möchten. Beispiel: **444455556666**.
9. Wählen Sie Speichern.

Sie können diese Richtlinie auch mit den folgenden add-permission AWS Command Line Interface (AWS CLI) -Befehlen erstellen. Wenn Sie die verwenden AWS CLI, müssen Sie die `lambda:InvokeFunction` Anweisungen `lambda:InvokeFunctionUrl` und separat hinzufügen. Beispiel:

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal 444455556666 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal 444455556666 \
  --invoked-via-function-url
```

Verwenden Sie den **`Authentifizierungstyp NONE`**

> **Important**
>
> Wenn Ihre Funktions-URL-Auth-Typ `NONE` ist und Sie eine ressourcenbasierte Richtlinie haben, die öffentlichen Zugriff gewährt, kann jeder nicht authentifizierte Benutzer mit Ihrer Funktions-URL Ihre Funktion aufrufen.

In einigen Fällen möchten Sie möglicherweise, dass Ihre Funktions-URL öffentlich ist. So können Sie beispielsweise Anfragen direkt von einem Webbrowser aus bereitstellen. Um den öffentlichen Zugriff auf Ihre Funktions-URL zu ermöglichen, wählen Sie den Auth-Typ `NONE` aus.

Wenn Sie den Auth-Typ `NONE` auswählen, verwendet Lambda IAM nicht, um Anfragen an Ihre Funktions-URL zu authentifizieren. Ihre Funktion muss jedoch über eine ressourcenbasierte Richtlinie verfügen, die `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` erlaubt. Wenn Sie `NONE` mithilfe der Konsole oder AWS Serverless Application Model (AWS SAM) eine Funktions-URL

mit dem Authentifizierungstyp erstellen, erstellt Lambda automatisch die ressourcenbasierte Richtlinie für Sie. Wenn Sie die AWS CLI AWS CloudFormation, oder die Lambda-API direkt verwenden, müssen Sie die Richtlinie selbst hinzufügen.

Wir empfehlen, dass Sie den InvokedViaFunctionUrl Kontextschlüssel lambda: in Ihre ressourcenbasierten Richtlinien aufnehmen, wenn Sie den Authentifizierungstyp verwenden. `NONE` Dieser Kontextschlüssel stellt sicher, dass die Funktion nur über die Funktions-URL und nicht über andere Aufrufmethoden aufgerufen werden kann.

Beachten Sie die folgenden Informationen über diese Richtlinie:

- Alle Entitäten können `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` aufrufen. Das bedeutet, dass jeder, der Ihre Funktions-URL hat, Ihre Funktion aufrufen kann.
- Der Wert des `lambda:FunctionUrlAuthType`-Bedingungsschlüssels ist `NONE`. Das bedeutet, dass diese Richtlinienanweisung den Zugriff nur erlaubt, wenn der Authentifizierungstyp Ihrer Funktions-URL ebenfalls `NONE` ist.
- Die Bedingung `lambda:InvokedViaFunctionUrl` stellt sicher, dass die Funktion nur über die Funktions-URL und nicht über andere Aufrufmethoden aufgerufen werden kann.

Example– Ressourcenbasierte Standardrichtlinie für den Authentifizierungstyp NONE

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "FunctionURLAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "NONE"
        }
      }
    },
    {
```

```
      "Sid": "FunctionURLInvokeAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

Erstellen Sie die ressourcenbasierte Richtlinie mithilfe des AWS CLI

Sofern Sie nicht die Konsole verwenden oder AWS SAM eine Funktions-URL mit Authentifizierungstyp erstellen`NONE`, müssen Sie die ressourcenbasierte Richtlinie selbst hinzufügen. Verwenden Sie die folgenden Befehle zur Erstellung von Anweisungen für die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction`. Jede Anweisung muss in einem separaten Befehl hinzugefügt werden.

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal * \
  --function-url-auth-type NONE
```

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal * \
  --invoked-via-function-url
```

> **Note**
>
> Wenn Sie Ihre Funktions-URL mit Authentifizierungstyp `NONE` löschen, löscht Lambda nicht automatisch die zugehörige ressourcenbasierte Richtlinie. Wenn Sie diese Richtlinie löschen möchten, müssen Sie dies manuell tun.

Wenn die ressourcenbasierte Richtlinie einer Funktion keine `lambda:InvokeFunction` Berechtigungen gewährt`lambda:invokeFunctionUrl`, erhalten Benutzer den Fehlercode 403 Forbidden, wenn sie versuchen, Ihre Funktions-URL aufzurufen. Dieser Fehler tritt auch dann auf, wenn die Funktions-URL den `NONE` Authentifizierungstyp verwendet.

Governance und Zugriffskontrolle

Zusätzlich zu den Berechtigungen für den URL-Aufruf von Funktionen können Sie auch den Zugriff auf Aktionen kontrollieren, die zum Konfigurieren von Funktions-URLs verwendet werden. Lambda unterstützt die folgenden IAM-Richtlinienaktionen für Funktions-URLs:

- `lambda:InvokeFunctionUrl`— Ruft eine Lambda-Event-Funktion mit der Funktions-URL auf.
- `lambda:CreateFunctionUrlConfig` – Erstellt eine Funktions-URL und legt ihren `AuthType` fest.
- `lambda:UpdateFunctionUrlConfig` – Aktualisiert eine Funktions-URL-Konfiguration und ihren `AuthType`.
- `lambda:GetFunctionUrlConfig` – Zeigt die Details einer Funktions-URL an.
- `lambda:ListFunctionUrlConfigs` – Listet die Funktions-URL-Konfigurationen auf.
- `lambda:DeleteFunctionUrlConfig` – Löscht eine Funktions-URL.

Um anderen AWS Entitäten den Zugriff auf Funktions-URLs zu gewähren oder zu verweigern, nehmen Sie diese Aktionen in die IAM-Richtlinien auf. Die folgende Richtlinie gewährt der `example` Rolle in beispielsweise AWS-Konto `444455556666` Berechtigungen zur Aktualisierung der Funktions-URL für die Funktion **`my-function`** im Konto`123456789012`.

Example Kontoübergreifende Funktions-URL-Richtlinie

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-
function"
        }
    ]
}
```

Bedingungsschlüssel

Verwenden Sie einen Bedingungskontextschlüssel, um eine feinabgestimmte Zugriffskontrolle über Ihre Funktions-URLs zu erhalten. Lambda unterstützt die folgenden Kontextschlüssel für Funktions-URLs.

- `lambda:FunctionUrlAuthType` – Definiert einen Aufzählungswert, der den Authentifizierungstyp beschreibt, den Ihre Funktions-URL verwendet. Der Wert kann entweder `AWS_IAM` oder `NONE` sein.
- `lambda:InvokedViaFunctionUrl` – Beschränkt die Aktion `lambda:InvokeFunction` auf Aufrufe, die über die Funktions-URL getätigt werden. Das stellt sicher, dass die Funktion nur über die Funktions-URL und nicht über andere Aufrufmethoden aufgerufen werden kann. Beispiele für ressourcenbasierte Richtlinien, die den Kontextschlüssel `lambda:InvokedViaFunctionUrl` verwenden, finden Sie in den Beispielen unter Verwenden des Authentifizierungstyps `AWS_IAM` und Verwenden Sie den Authentifizierungstyp `NONE`.

Sie können diese Kontextschlüssel in Richtlinien verwenden, die mit Ihrer Event-Funktion verknüpft sind. Sie können beispielsweise einschränken, wer Konfigurationsänderungen an Ihren Funktions-URLs vornehmen kann. Um alle `UpdateFunctionUrlConfig`-Anfragen an jede Funktion mit dem URL-Auth-Typ `NONE` zu verweigern, können Sie die folgende Richtlinie definieren:

Example Funktions-URL-Richtlinie mit expliziter Zugriffsverweigerung

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

Um der `example` Rolle bei Funktionen mit URL-Authentifizierungstyp AWS-Konto `444455556666` Berechtigungen `CreateFunctionUrlConfig` und `UpdateFunctionUrlConfig` Anfragen zu erteilen`AWS_IAM`, können Sie die folgende Richtlinie definieren:

Example Funktions-URL-Richtlinie mit expliziter Zulassung

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
```

```
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

Sie können diesen Bedingungsschlüssel auch in einer Service-Kontrollrichtlinie (SCP) verwenden. Verwenden Sie SCPs, um Berechtigungen in einer gesamten Organisation in AWS Organizations zu verwalten. Zum Beispiel, um Benutzern das Erstellen oder Aktualisieren von Funktions-URLs zu verweigern, die etwas anderes als den Auth-Typ `AWS_IAM` verwenden, verwenden Sie die folgende Service-Kontrollrichtlinie:

Example Funktions-URL-SCP mit expliziter Zugriffsverweigerung

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
```

```
}
```

Aufrufen von Lambda-Funktions-URLs

Eine Funktions-URL ist ein dedizierter HTTP (S) -Endpunkt für Ihre Lambda-Ereignisfunktion. Sie können eine Funktions-URL über die Lambda-Konsole oder die Lambda-API erstellen und konfigurieren.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: Funktions-URLs und Amazon API Gateway. Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “Funktions-URLs im Vergleich zu Amazon API Gateway”.

Wenn Sie eine Funktions-URL erstellen, generiert Lambda automatisch einen eindeutigen URL-Endpunkt für Sie. Sobald Sie eine Funktions-URL erstellt haben, ändert sich ihr URL-Endpunkt nie mehr. Funktions-URL-Endpunkte haben das folgende Format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Funktions-URLs werden nicht in allen AWS Regionen unterstützt. Informationen zur Verfügbarkeit finden Sie unter AWS Regionaldienste und filtern Sie nach`Feature Name = Function URLs`.

Funktions-URLs sind Dual-Stack-fähig und unterstützen IPv4 und IPv6. Nachdem Sie Ihre Funktions-URL konfiguriert haben, können Sie Ihre Funktion über ihren HTTP (S) -Endpunkt aufrufen, indem Sie einen Webbrowser, Curl, Postman oder einen beliebigen HTTP-Client verwenden. Um eine Funktions-URL aufzurufen, müssen Sie `lambda:InvokeFunctionUrl`- und `lambda:InvokeFunction`-Berechtigungen haben. Weitere Informationen finden Sie unter Zugriffskontrolle.

Themen

- Grundlagen des Aufrufs von Funktions-URLs
- Anfordern und Beantworten von Nutzlasten

Grundlagen des Aufrufs von Funktions-URLs

Wenn Ihre Funktions-URL den Auth-Typ `AWS_IAM` hat, müssen Sie jede HTTP-Anfrage mit der AWS Signature Version 4 (SigV4) signieren. Tools wie Postman bieten integrierte Möglichkeiten, Ihre Anforderungen mit SigV4 zu signieren.

Wenn Sie kein Tool verwenden, um HTTP-Anfragen an Ihre Funktions-URL zu signieren, müssen Sie jede Anfrage manuell mit SigV4 signieren. Wenn Ihre Funktions-URL eine Anfrage erhält, berechnet Lambda auch die SigV4-Signatur. Nur wenn die Signaturen übereinstimmen, verarbeitet Lambda die Anfrage. Anweisungen zum manuellen Signieren Ihrer Anfragen mit Sigv4 finden Sie unter Signieren von AWS Anfragen mit Signature Version 4 im Handbuch.Allgemeine Amazon Web Services-Referenz

Wenn Ihre Funktions-URL den Authentifizierungstyp `NONE` verwendet, müssen Sie Ihre Anfragen nicht mit SigV4 signieren. Sie können Ihre Funktion per Webbrowser, cURL, Postman oder einem beliebigen HTTP-Client aufrufen.

Um einfache `GET`-Anfragen an Ihre Funktion zu testen, verwenden Sie einen Webbrowser. Wenn Ihre Funktions-URL zum Beispiel `https://abcdefg.lambda-url.us-east-1.on.aws` ist und sie einen Zeichenfolgeparameter `message` aufnimmt, könnte Ihre Anforderungs-URL folgendermaßen aussehen:

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

Um andere HTTP-Anfragen zu testen, z. B. eine `POST`-Anfrage, können Sie ein Tool wie cURL verwenden. Wenn Sie z. B. einige JSON-Daten in eine `POST`-Anfrage an Ihre Funktions-URL aufnehmen möchten, können Sie den folgenden cURL-Befehl verwenden:

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

Anfordern und Beantworten von Nutzlasten

Wenn ein Client Ihre Funktions-URL aufruft, ordnet Lambda die Anforderung einem Ereignisobjekt zu, bevor es sie an Ihre Funktion übergibt. Die Antwort Ihrer Funktion wird dann einer HTTP-Antwort zugeordnet, die Lambda über die Funktions-URL an den Client zurücksendet.

Die Formate für Anforderungs- und Antwortereignisse folgen dem gleichen Schema wie die Amazon-API-Gateway-Nutzlastformatversion 2.0.

Anforderungsnutzlastformat

Eine Anforderungsnutzlast hat die folgende Struktur:

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
```

```
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
  "isBase64Encoded": false,
  "stageVariables": null
}
```

Parameter	Description	Beispiel
`version`	Die Nutzlastformatversion für dieses Ereignis. Lambda-Funktions-URLs unterstützen derzeit die Nutzlastformatversion 2.0.	`2.0`
`routeKey`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `$default` als Platzhalter.	`$default`
`rawPath`	Der Anforderungspfad. Wenn die Anforderungs-URL beispielsweise `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` ist, dann ist der Rohpfadwert `/example/test/demo` .	`/example/test/demo`
`rawQueryString`	Die Rohzeichenfolge, die die Abfragezeichenfolge-Parameter der Anforderung enthält. Zu den unterstützten Zeichen	`"?parameter1=value1&parameter2=value2"`

Parameter	Description	Beispiel
	gehören `a-z`, `A-Z`, `0-9`, `.`, `_`, `-`, `%`, &, =, und `+`.	
`cookies`	Ein Array, das alle Cookies enthält, die im Rahmen der Anforderung gesendet wurden.	`["Cookie_1=Value_1", "Cookie_2=Value_2"]`
`headers`	Die Liste der Anforderungs-Header, die als Schlüssel -Wert-Paare dargestellt werden.	`{"header1": "value1", "header2": "value2"}`
`queryStringParameters`	Die Abfrageparameter für die Anforderung. Wenn die Anforderungs-URL beispielsweise `https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane` ist, dann ist der `queryStringParameters` -Wert ein JSON-Objekt mit einem Schlüssel von `name` und einem Wert von `Jane`.	`{"name": "Jane"}`
`requestContext`	Ein Objekt, das zusätzliche Informationen über die Anforderung enthält, z. B. `requestId` , den Zeitpunkt der Anfrage und die Identität des Anrufers, falls über AWS Identity and Access Management (IAM) autorisiert.	

Parameter	Description	Beispiel
`requestContext.accountId`	Die AWS-Konto ID des Funktionseigentümers.	`"123456789012"`
`requestContext.apiId`	Die ID der Funktions-URL.	`"33anwqw8fj"`
`requestContext.authentication`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null`.	`null`
`requestContext.authorizer`	Ein Objekt, das Informationen über die Aufruferidentität enthält, wenn die Funktions-URL den Authentifizierungstyp `AWS_IAM` verwendet. Ansonsten setzt Lambda dies auf `null`.	
`requestContext.authorizer.iam.accessKey`	Der Zugriffsschlüssel der Anruferidentität.	`"AKIAIOSFODNN7EXAMPLE"`
`requestContext.authorizer.iam.accountId`	Die AWS-Konto ID der Anruferidentität.	`"111122223333"`
`requestContext.authorizer.iam.callerId`	Die ID (Benutzer-ID) des Aufrufers.	`"AIDACKCEVSQ6C2EXAMPLE"`
`requestContext.authorizer.iam.cognitoIdentity`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null` oder schließt dies aus der JSON aus.	`null`

Parameter	Description	Beispiel
`requestContext.authorizer.iam.principalOrgId`	Die Prinzipal-Organisations-ID, die mit der Anruferidentität verknüpft ist.	`"AIDACKCEVSQORGEXAMPLE"`
`requestContext.authorizer.iam.userArn`	Der Benutzer-ARN (Amazon-Ressourcenname) der Anruferidentität.	`"arn:aws:iam::111122223333:user/example-user"`
`requestContext.authorizer.iam.userId`	Die Benutzer-ID des Anrufers.	`"AIDACOSFODNN7EXAMPLE2"`
`requestContext.domainName`	Der Domain-Name der Funktions-URL.	`"<url-id>.lambda-url.us-west-2.on.aws"`
`requestContext.domainPrefix`	Das Domain-Präfix der Funktions-URL.	`"<url-id>"`
`requestContext.http`	Ein Objekt, das Details zur HTTP-Anforderung enthält.	
`requestContext.http.method`	Die in der Anforderung verwendete HTTP-Methode. Gültige Werte sind unter anderem `GET`, `POST`, `PUT`, `HEAD`, `OPTIONS`, `PATCH` und `DELETE`.	`GET`
`requestContext.http.path`	Der Anforderungspfad. Zum Beispiel, wenn die Anforderungs-URL `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` ist, dann ist der Pfadwert `/example/test/demo` .	`/example/test/demo`

Parameter	Description	Beispiel
`requestContext.http.protocol`	Das Protokoll der Anforderung.	`HTTP/1.1`
`requestContext.http.sourceIp`	Die Quell-IP-Adresse der TCP-Verbindung, von der die Anforderung stammt.	`123.123.123.123`
`requestContext.http.userAgent`	Der Header-Wert der User-Agent Anfrage.	`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0`
`requestContext.requestId`	Die ID der aktuellen Aufrufanforderung. Sie können diese ID verwenden, um Aufrufprotokolle zu verfolgen, die sich auf Ihre Funktion beziehen.	`e1506fd5-9e7b-434f-bd42-4f8fa224b599`
`requestContext.routeKey`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `$default` als Platzhalter.	`$default`
`requestContext.stage`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `$default` als Platzhalter.	`$default`
`requestContext.time`	Der Zeitstempel der Anfrage.	`"07/Sep/2021:22:50:22 +0000"`
`requestContext.timeEpoch`	Die Uhrzeit der Anfrage in Unix-Epochen-Zeit.	`"1631055022677"`

Parameter	Description	Beispiel
body	Der Text der Anforderung. Wenn der Inhaltstyp der Anforderung binär ist, ist der Text base64-kodiert.	`{"key1": "value1", "key2": "value2"}`
`pathParameters`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null` oder schließt dies aus der JSON aus.	`null`
`isBase64Encoded`	`TRUE`, wenn der Text eine binäre Nutzlast hat und base64-kodiert ist. Ansonsten `FALSE`.	`FALSE`
`stageVariables`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null` oder schließt dies aus der JSON aus.	`null`

Antwortnutzlastformat

Wenn Ihre Funktion eine Antwort zurückgibt, analysiert Lambda die Antwort und konvertiert sie in eine HTTP-Antwort. Funktionsantwortnutzlasten haben das folgende Format:

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
```

```
    ],
    "isBase64Encoded": false
}
```

Lambda leitet das Antwortformat für Sie ab. Wenn Ihre Funktion gültiges JSON zurückgibt und keinen `statusCode` zurückgibt, geht Lambda von Folgendem aus:

- `statusCode` ist `200`.

> **Note**
>
> Die gültigen `statusCode` liegen im Bereich von 100 bis 599.

- `content-type` ist `application/json`.
- `body` ist die Antwort der Funktion.
- `isBase64Encoded` ist `false`.

Die folgenden Beispiele zeigen, wie die Ausgabe Ihrer Lambda-Ereignisfunktion der Antwortnutzlast zugeordnet wird und wie die Antwortnutzlast der endgültigen HTTP-Antwort zugeordnet wird. Wenn der Client Ihre Funktions-URL aufruft, wird die HTTP-Antwort angezeigt.

Beispielausgabe für eine Zeichenfolgeantwort

Ausgabe der Lambda-Ereignisfunktion	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)

`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 15`
		
		`"Hello, world!"`

Beispielausgabe für eine JSON-Antwort

Ausgabe der Lambda-Ereignisfunktion	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
`{ "message": "Hello, world!" }`	`{ "statusCode": 200, "body": { "message": "Hello, world!" }, "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200 date: Wed, 08 Sep 2021 18:02:24 GMT content-type: application/json content-length: 34 { "message": "Hello, world!" }`

Beispielausgabe für eine benutzerdefinierte Antwort

Ausgabe der Lambda-Ereignisfunktion	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
`{ "statusCode": 201, "headers": { "Content-Type": "application/json", "My-Custom-Header": "Custom Value" }, "body": JSON.stringify({ "message": "Hello, world!" }), "isBase64Encoded": false`	`{ "statusCode": 201, "headers": { "Content-Type": "application/json", "My-Custom-Header": "Custom Value" }, "body": JSON.stringify({ "message": "Hello, world!" }), "isBase64Encoded": false`	`HTTP/2 201 date: Wed, 08 Sep 2021 18:02:24 GMT content-type: application/json content-length: 27 my-custom-header: Custom Value { "message": "Hello, world!" }`

Ausgabe der Lambda-Ereignisfunktion	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
`}`	`}`	

Cookies

Um Cookies von Ihrer Funktion zurückzugeben, fügen Sie `set-cookie`-Header nicht manuell hinzu. Fügen Sie stattdessen die Cookies in Ihr Antwortnutzlastobjekt ein. Lambda interpretiert dies automatisch und fügt sie als `set-cookie`-Header in Ihre HTTP-Antwort ein, wie im folgenden Beispiel gezeigt.

Ausgabe der Lambda-Ereignisfunktion	HTTP-Antwort (was der Client sieht)

`{`
`  "statusCode": 201,`
`  "headers": {`
`  "Content-Type": "application/json",`
`  "My-Custom-Header": "Custom Value"`
`  },`
`  "body": JSON.stringify({`
`  "message": "Hello, world!"`
`  }),`
`  "cookies": [`
`  "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",`
`  "Cookie_2=Value2; Max-Age=78000"`
`  ],`
`  "isBase64Encoded": false`
`}`
	`HTTP/2 201`
	`date: Wed, 08 Sep 2021 18:02:24 GMT`
	`content-type: application/json`
	`content-length: 27`
	`my-custom-header: Custom Value`
	`set-cookie: Cookie_1=Value2; Expires=21 Oct 2021 07:48 GMT`
	`set-cookie: Cookie_2=Value2; Max-Age=78000`
	
	`{`
	`  "message": "Hello, world!"`
	`}`

Überwachen der Lambda-Funktions-URLs

Sie können AWS CloudTrail und Amazon CloudWatch zur Überwachung Ihrer Funktions-URLs verwenden.

Themen

- Überwachen von Funktions-URLs mit CloudTrail
- CloudWatch-Metriken für Funktions-URLs

Überwachen von Funktions-URLs mit CloudTrail

Bei Funktions-URLs unterstützt Lambda automatisch die Protokollierung der folgenden API-Operationen als Ereignisse in CloudTrail-Protokolldateien:

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

Jeder Protokolleintrag enthält Informationen über die Anruferidentität, wann die Anfrage gestellt wurde, und andere Details. Sie können alle Ereignisse innerhalb der letzten 90 Tage sehen, indem Sie Ihren CloudTrail-Ereignisverlauf ansehen. Um Datensätze nach 90 Tagen beizubehalten, können Sie einen Trail erstellen.

Standardmäßig protokolliert CloudTrail `InvokeFunctionUrl`-Anfragen, die für Datenereignisse gehalten werden, nicht. Sie können die Datenereignisprotokollierung jedoch in CloudTrail aktivieren. Weitere Informationen finden Sie unter Protokollieren von Datenereignissen für Trails im AWS CloudTrail-Benutzerhandbuch.

CloudWatch-Metriken für Funktions-URLs

Lambda sendet aggregierte Metriken über Funktions-URL-Anfragen an CloudWatch. Mit diesen Metriken können Sie Ihre Funktions-URLs überwachen, Dashboards erstellen und Alarme in der CloudWatch-Konsole konfigurieren.

Funktions-URLs unterstützen die folgenden Aufrufmetriken. Wir empfehlen, diese Metriken mit der `Sum`-Statistik anzuzeigen.

- `UrlRequestCount` – Die Anzahl der Anforderungen an diese Funktions-URL.
- `Url4xxCount` – Die Anzahl der Anforderungen, bei denen ein 4XX-HTTP-Statuscode zurückgegeben wurde. Codes der 4XX-Serie weisen auf clientseitige Fehler hin, z. B. schlechte Anforderungen.
- `Url5xxCount` – Die Anzahl der Anforderungen, bei denen ein 5XX-HTTP-Statuscode zurückgegeben wurde. Codes der 5XX-Serie weisen auf serverseitige Fehler wie Funktionsfehler und Timeouts hin.

Funktions-URLs unterstützen auch die folgende Leistungsmetrik. Wir empfehlen, diese Metriken mit der `Average`- oder `Max`-Statistik anzuzeigen.

- `UrlRequestLatency` – Die Zeit zwischen dem Erhalt der Funktions-URL eine Anforderung und wann die Funktions-URL eine Antwort zurückgibt.

Jede dieser Aufrufs- und Leistungsmetriken unterstützt die folgenden Dimensionen:

- `FunctionName` – Zeigt Aggregatmetriken für Funktions-URLs an, die der `$LATEST` unveröffentlichten Version oder einem der Aliase der Funktion zugeordnet sind. Beispiel, `hello-world-function`.
- `Resource` – Zeigt Metriken für eine bestimmte Funktions-URL an. Dies wird durch einen Funktionsnamen zusammen mit der `$LATEST` unveröffentlichten Version oder einem der Aliase der Funktion definiert. Beispiel, `hello-world-function:$LATEST`.
- `ExecutedVersion` – Zeigt Metriken für eine bestimmte Funktions-URL basierend auf der ausgeführten Version an. Sie können diese Dimension hauptsächlich verwenden, um die Funktions-URL zu verfolgen, die der `$LATEST`unveröffentlichten Version zugeordnet ist.

Wählen Sie eine Methode aus, um Ihre Lambda-Ereignisfunktion mithilfe einer HTTP-Anfrage aufzurufen

In vielen gängigen Anwendungsfällen für Lambda wird Ihre Event-Funktion mithilfe einer HTTP-Anfrage aufgerufen. Beispielsweise möchten Sie möglicherweise, dass eine Webanwendung Ihre Funktion über eine Browseranforderung aufruft. Lambda Event Functions können auch verwendet werden, um vollständige REST-APIs zu erstellen, Benutzerinteraktionen aus mobilen Apps zu verarbeiten, Daten von externen Diensten über HTTP-Aufrufe zu verarbeiten oder benutzerdefinierte Webhooks zu erstellen.

In den folgenden Abschnitten wird erläutert, welche Möglichkeiten Sie haben, Lambda über HTTP aufzurufen und es werden Informationen bereitgestellt, die Ihnen helfen, die richtige Entscheidung für Ihren speziellen Anwendungsfall zu treffen.

Welche Optionen stehen Ihnen bei der Auswahl einer HTTP-Aufrufmethode zur Verfügung?

Lambda bietet zwei Hauptmethoden zum Aufrufen einer Funktion mithilfe einer HTTP-Anfrage: Funktions-URLs und API-Gateway. Die wichtigsten Unterschiede zwischen diesen beiden Optionen sind die folgenden:

- Lambda-Funktions-URLs bieten einen einfachen, direkten HTTP-Endpunkt für eine Lambda-Ereignisfunktion. Sie sind auf Einfachheit und Wirtschaftlichkeit optimiert und bieten den schnellsten Weg, eine Lambda-Ereignisfunktion über HTTP verfügbar zu machen.
- API Gateway ist ein fortschrittlicherer Dienst zum Erstellen von APIs mit vollem Funktionsumfang. API Gateway ist für die Erstellung und Verwaltung von Produktions-APIs in großem Maßstab optimiert und bietet umfassende Tools für Sicherheit, Überwachung und Datenverkehrsmanagement.

Empfehlungen, falls Sie Ihre Anforderungen bereits kennen

Wenn Sie sich über Ihre Anforderungen bereits im Klaren sind, finden Sie hier unsere grundlegenden Empfehlungen:

Wir empfehlen Funktions-URLs für einfache Anwendungen oder das Prototyping, bei denen Sie nur grundlegende Authentifizierungsmethoden und deren request/response Handhabung benötigen und bei denen Sie Kosten und Komplexität auf ein Minimum reduzieren möchten.

API Gateway ist eine bessere Wahl für Produktionsanwendungen in großem Maßstab oder für Fälle, in denen Sie erweiterte Funktionen wie OpenAPI-Description, eine Auswahl an Authentifizierungsoptionen, benutzerdefinierte Domainnamen oder umfangreiche request/response Funktionen wie Drosselung, Caching und Transformation benötigen. request/response

Was Sie bei der Auswahl einer Methode zum Aufrufen Ihrer Lambda-Ereignisfunktion beachten sollten

Bei der Wahl zwischen Funktions-URLs und API-Gateway müssen Sie die folgenden Faktoren berücksichtigen:

- Ihre Authentifizierungsanforderungen, z. B. ob Sie OAuth oder Amazon Cognito zur Benutzerauthentifizierung benötigen

- Ihre Skalierungsanforderungen und die Komplexität der API, die Sie implementieren möchten
- Ob Sie erweiterte Funktionen wie die Validierung und Formatierung von Anfragen benötigen request/response
- Ihre Überwachungsanforderungen
- Was sind Ihre Ziele?

Wenn Sie diese Faktoren verstehen, können Sie die Option auswählen, die Ihren Anforderungen an Sicherheit, Komplexität und Kosten am besten gerecht wird.

Die folgenden Informationen fassen die wichtigsten Unterschiede zwischen den beiden Optionen zusammen.

Authentifizierung

- Funktions-URLs bieten grundlegende Authentifizierungsoptionen über AWS Identity and Access Management (IAM). Sie können Ihre Endpunkte so konfigurieren, dass sie entweder öffentlich sind (keine Authentifizierung) oder dass sie eine IAM-Authentifizierung erfordern. Mit der IAM-Authentifizierung können Sie AWS Standardanmeldedaten oder IAM-Rollen verwenden, um den Zugriff zu steuern. Dieser Ansatz ist zwar einfach einzurichten, bietet aber im Vergleich zu anderen Authentifizierungsmethoden nur begrenzte Optionen.
- API Gateway bietet Zugriff auf eine umfassendere Palette von Authentifizierungsoptionen. Neben der IAM-Authentifizierung können Sie Lambda-Autorisierer (benutzerdefinierte Authentifizierungslogik), Amazon Cognito Cognito-Benutzerpools und Flows verwenden. OAuth2.0 Diese Flexibilität ermöglicht es Ihnen, komplexe Authentifizierungsschemata zu implementieren, darunter Authentifizierungsanbieter von Drittanbietern, tokenbasierte Authentifizierung und Multi-Faktor-Authentifizierung.

Request/response Handhabung

- Funktions-URLs ermöglichen die grundlegende Behandlung von HTTP-Anfragen und -Antworten. Sie unterstützen Standard-HTTP-Methoden und bieten integrierte CORS-Unterstützung (Cross-Origin Resource Sharing). Sie können zwar JSON-Nutzdaten und Abfrageparameter auf natürliche Weise verarbeiten, bieten jedoch keine Funktionen zur Transformation oder Validierung von Anfragen. Die Bearbeitung von Antworten ist ähnlich einfach — der Client erhält die Antwort von Ihrer Lambda-Ereignisfunktion genau so, wie Lambda sie zurückgibt.

- API Gateway bietet ausgefeilte Funktionen zur Bearbeitung von Anfragen und Antworten. Sie können Anforderungsvalidatoren definieren, Anfragen und Antworten mithilfe von Zuordnungsvorlagen transformieren, request/response Header einrichten und das Zwischenspeichern von Antworten implementieren. API Gateway unterstützt auch binäre Nutzdaten und benutzerdefinierte Domainnamen und kann Antworten ändern, bevor sie den Client erreichen. Sie können Modelle für die request/response Validierung und Transformation mithilfe des JSON-Schemas einrichten.

Skalierung

- Funktions-URLs passen sich direkt an die Parallelitätsgrenzen Ihrer Lambda-Ereignisfunktion an und bewältigen Verkehrsspitzen, indem Sie Ihre Funktion bis zum konfigurierten Maximallimit skalieren. Sobald dieses Limit erreicht ist, reagiert Lambda auf weitere Anfragen mit HTTP 429-Antworten. Es gibt keinen integrierten Warteschlangenmechanismus, sodass der Umgang mit der Skalierung vollständig von der Konfiguration Ihrer Lambda-Ereignisfunktion abhängt. Standardmäßig haben Lambda-Ereignisfunktionen ein Limit von 1.000 gleichzeitigen Ausführungen pro. AWS-Region
- API Gateway bietet zusätzlich zur Lambda-eigenen Skalierung zusätzliche Skalierungsfunktionen. Es umfasst integrierte Funktionen zur Warteschleife und Drosselung von Anfragen, sodass Sie Datenverkehrsspitzen besser bewältigen können. API Gateway kann standardmäßig bis zu 10.000 Anfragen pro Sekunde pro Region verarbeiten, mit einer Burst-Kapazität von 5.000 Anfragen pro Sekunde. Es bietet auch Tools zum Drosseln von Anfragen auf verschiedenen Ebenen (API, Phase oder Methode), um Ihr Backend zu schützen.

Überwachen

- Funktions-URLs ermöglichen eine grundlegende Überwachung anhand von CloudWatch Amazon-Metriken, einschließlich Anzahl der Anfragen, Latenz und Fehlerraten. Sie erhalten Zugriff auf standardmäßige Lambda-Metriken und -Protokolle, die die Rohanfragen anzeigen, die in Ihre Funktion eingehen. Dies bietet zwar wichtige betriebliche Transparenz, die Metriken konzentrieren sich jedoch hauptsächlich auf die Funktionsausführung.
- API Gateway bietet umfassende Überwachungsfunktionen, einschließlich detaillierter Metriken, Protokollierungs- und Verfolgungsoptionen. Sie können API-Aufrufe, Latenz, Fehlerraten und hit/miss Cache-Raten über überwachen CloudWatch. API Gateway lässt sich auch AWS X-Ray für verteiltes Tracing integrieren und bietet anpassbare Protokollierungsformate.

Cost (Kosten)

- Funktions-URLs folgen dem Standard-Lambda-Preismodell – Sie zahlen nur für Funktionsaufrufe und Rechenzeit. Für den URL-Endpunkt selbst fallen keine zusätzlichen Gebühren an. Dies macht es zu einer kostengünstigen Wahl für einfache APIs oder Anwendungen mit geringem Datenverkehr, wenn Sie die zusätzlichen Features von API Gateway nicht benötigen.
- API Gateway bietet ein kostenloses Kontingent, das eine Million API-Aufrufe für REST-APIs und eine Million API-Aufrufe für HTTP-APIs umfasst. Danach berechnet API Gateway Gebühren für API-Aufrufe, Datenübertragung und Caching (falls aktiviert). Weitere Informationen zu den Kosten für Ihren eigenen Anwendungsfall finden Sie auf der Seite mit den Preisen für API Gateway.

Weitere Funktionen

- Funktions-URLs sind auf Einfachheit und direkte Lambda-Integration ausgelegt. Sie unterstützen sowohl HTTP- als auch HTTPS-Endpunkte, bieten integrierte CORS-Unterstützung und bieten Dual-Stack-Endpunkte (IPv4 und IPv6). Ihnen fehlen zwar erweiterte Funktionen, sie zeichnen sich jedoch in Szenarien aus, in denen Sie Lambda-Ereignisfunktionen schnell und unkompliziert über HTTP verfügbar machen müssen.
- API Gateway umfasst zahlreiche zusätzliche Funktionen wie API-Versionierung, Stufenmanagement, API-Schlüssel für Nutzungspläne, API-Dokumentation über Swagger/OpenAPI, WebSocket APIs, private APIs innerhalb einer VPC und WAF-Integration für zusätzliche Sicherheit. Es unterstützt auch kanarische Implementierungen, Scheinintegrationen zum Testen und die Integration mit anderen Systemen AWS-Services als Lambda.

Wählen Sie eine Methode zum Aufrufen Ihrer Lambda-Ereignisfunktion

Nachdem Sie sich mit den Kriterien für die Auswahl zwischen Lambda-Funktions-URLs und API Gateway und den wichtigsten Unterschieden zwischen ihnen vertraut gemacht haben, können Sie die Option auswählen, die Ihren Anforderungen am besten entspricht und die folgenden Ressourcen verwenden, um Ihnen den Einstieg zu erleichtern.

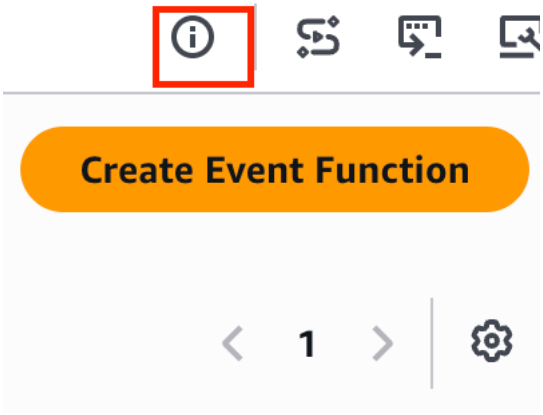
Function URLs

Nutzen Sie die folgenden Ressourcen, um mit Funktions-URLs zu beginnen

- Folgen Sie dem Tutorial Erstellen einer Lambda-Ereignisfunktion mit einer Funktions-URL

- Weitere Informationen zu Funktions-URLs finden Sie im the section called “Funktions-URLs”-Kapitel dieses Handbuchs
- Probieren Sie das in der Konsole enthaltene Tutorial Erstellen Sie eine einfache Web-App aus, indem Sie wie folgt vorgehen:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Öffnen Sie das Hilfe-Panel, indem Sie das Symbol in der oberen rechten Ecke des Bildschirms wählen.



3. Wählen Sie Tutorials aus.
4. Wählen Sie unter Einfache Web-App erstellen die Option Tutorial starten aus.

API Gateway

Beginnen Sie mit Lambda und API Gateway mit den folgenden Ressourcen

- Folgen Sie dem Tutorial Verwenden von Lambda mit API Gateway, um eine REST-API zu erstellen, die in eine Lambda-Ereignisfunktion im Backend integriert ist.
- In den folgenden Abschnitten des Amazon API Gateway Developer Guide erfahren Sie mehr über die verschiedenen Arten von APIs, die von API Gateway angeboten werden:
  - API Gateway-REST-APIs
  - API Gateway-HTTP-APIs
  - WebSocket API-Gateway-APIs
- Probieren Sie eines oder mehrere der Beispiele im Abschnitt Tutorials und Workshops im Amazon API Gateway Developer Guide aus.

Tutorial: Erstellen eines Webhook-Endpunkts mithilfe einer Lambda-Funktions-URL

In diesem Tutorial erstellen Sie eine Lambda-Funktions-URL, um einen Webhook-Endpunkt zu implementieren. Ein Webhook ist eine einfache, ereignisgesteuerte Kommunikation, die automatisch Daten zwischen Anwendungen über HTTP sendet. Sie können Webhooks verwenden, um sofortige Benachrichtigungen über Ereignisse in einem anderen System zu erhalten, z. B. wenn sich ein neuer Kunde auf einer Website registriert, eine Zahlung verarbeitet wird oder eine Datei hochgeladen wird.

Mit Lambda können Webhooks entweder über Lambda-Funktions-URLs oder API Gateway implementiert werden. Funktions-URLs sind eine gute Wahl für einfache Webhooks, die keine Funktionen wie erweiterte Autorisierung oder Anforderungsvalidierung erfordern.

> **Tip**
>
> Wenn Sie sich nicht sicher sind, welche Lösung für Ihren speziellen Anwendungsfall am besten geeignet ist, lesen Sie bitte the section called “Funktions-URLs im Vergleich zu Amazon API Gateway”.

Voraussetzungen

Um dieses Tutorial abzuschließen, müssen Sie entweder Python (Version 3.8 oder höher) oder Node.js (Version 18 oder höher) auf Ihrem lokalen Computer installiert haben.

Um den Endpunkt mit einer HTTP-Anfrage zu testen, verwendet das Tutorial curl, ein Befehlszeilentool, mit dem Sie Daten über verschiedene Netzwerkprotokolle übertragen können. In der curl-Dokumentation erfahren Sie, wie Sie das Tool installieren, falls Sie das noch nicht getan haben.

Erstellen Sie die Lambda-Ereignisfunktion

Erstellen Sie zunächst die Lambda-Ereignisfunktion, die ausgeführt wird, wenn eine HTTP-Anfrage an Ihren Webhook-Endpunkt gesendet wird. In diesem Beispiel sendet die sendende Anwendung bei jeder Zahlung eine Aktualisierung und gibt im Hauptteil der HTTP-Anfrage an, ob die Zahlung erfolgreich war. Die Lambda-Ereignisfunktion analysiert die Anfrage und ergreift entsprechend dem Status der Zahlung Maßnahmen. In diesem Beispiel gibt der Code lediglich die Bestellnummer für die Zahlung aus, aber in einer realen Anwendung würden Sie die Bestellung möglicherweise einer Datenbank hinzufügen oder eine Benachrichtigung senden.

Die Funktion implementiert auch die gängigste Authentifizierungsmethode für Webhooks, die hashbasierte Nachrichtenauthentifizierung (HMAC). Bei dieser Methode teilen sich sowohl die sendende als auch die empfangende Anwendung einen geheimen Schlüssel. Die sendende Anwendung verwendet einen Hash-Algorithmus, um anhand dieses Schlüssels und des Nachrichteninhalts eine eindeutige Signatur zu generieren, und fügt die Signatur als HTTP-Header in die Webhook-Anfrage ein. Die empfangende Anwendung wiederholt diesen Schritt, generiert die Signatur unter Verwendung des geheimen Schlüssels und vergleicht den resultierenden Wert mit der im Anfrage-Header gesendeten Signatur. Stimmt das Ergebnis überein, wird die Anfrage als legitim angesehen.

Erstellen Sie die Funktion mithilfe der Lambda-Konsole entweder mit Python oder Node.js Runtime.

Python

Erstellen Sie die Lambda-Ereignisfunktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Erstellen Sie eine einfache „Hello World“-Funktion:
   a. Wählen Sie „Event-Funktion erstellen“.
   b. Wählen Sie Verfassen von Grund auf aus.
   c. Geben Sie für Function name (Funktionsname) **`myLambdaWebhook`** ein.
   d. Wählen Sie für Runtime python3.14 aus.
   e. Wählen Sie „Event-Funktion erstellen“.
3. Ersetzen Sie im Bereich Codequelle den vorhandenen Code durch Folgendes:

```
import json
import hmac
import hashlib
import os

def lambda_handler(event, context):

    # Get the webhook secret from environment variables
    webhook_secret = os.environ['WEBHOOK_SECRET']

    # Verify the webhook signature
    if not verify_signature(event, webhook_secret):
        return {
```

```
            'statusCode': 401,
            'body': json.dumps({'error': 'Invalid signature'})
        }

    try:
        # Parse the webhook payload
        payload = json.loads(event['body'])

        # Handle different event types
        event_type = payload.get('type')

        if event_type == 'payment.success':
            # Handle successful payment
            order_id = payload.get('orderId')
            print(f"Processing successful payment for order {order_id}")

            # Add your business logic here
            # For example, update database, send notifications, etc.

        elif event_type == 'payment.failed':
            # Handle failed payment
            order_id = payload.get('orderId')
            print(f"Processing failed payment for order {order_id}")

            # Add your business logic here

        else:
            print(f"Received unhandled event type: {event_type}")

        # Return success response
        return {
            'statusCode': 200,
            'body': json.dumps({'received': True})
        }

    except json.JSONDecodeError:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': 'Invalid JSON payload'})
        }
    except Exception as e:
        print(f"Error processing webhook: {e}")
        return {
            'statusCode': 500,
```

```
            'body': json.dumps({'error': 'Internal server error'})
        }

def verify_signature(event, webhook_secret):
    """
    Verify the webhook signature using HMAC
    """
    try:
        # Get the signature from headers
        signature = event['headers'].get('x-webhook-signature')

        if not signature:
            print("Error: Missing webhook signature in headers")
            return False

        # Get the raw body (return an empty string if the body key doesn't
 exist)
        body = event.get('body', '')

        # Create HMAC using the secret key
        expected_signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

        # Compare the expected signature with the received signature to
 authenticate the message
        is_valid = hmac.compare_digest(signature, expected_signature)
        if not is_valid:
            print(f"Error: Invalid signature. Received: {signature}, Expected:
 {expected_signature}")
            return False

        return True
    except Exception as e:
        print(f"Error verifying signature: {e}")
        return False
```

4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren.

Node.js

Erstellen Sie die Lambda-Ereignisfunktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Erstellen Sie eine einfache „Hello World“-Funktion:
   a. Wählen Sie „Event-Funktion erstellen“.
   b. Wählen Sie Verfassen von Grund auf aus.
   c. Geben Sie für Function name (Funktionsname) **myLambdaWebhook** ein.
   d. Wählen Sie für Runtime nodejs24.x aus.
   e. Wählen Sie „Event-Funktion erstellen“.
3. Ersetzen Sie im Bereich Codequelle den vorhandenen Code durch Folgendes:

```
import crypto from 'crypto';

export const handler = async (event, context) => {
    // Get the webhook secret from environment variables
    const webhookSecret = process.env.WEBHOOK_SECRET;

    // Verify the webhook signature
    if (!verifySignature(event, webhookSecret)) {
        return {
            statusCode: 401,
            body: JSON.stringify({ error: 'Invalid signature' })
        };
    }

    try {
        // Parse the webhook payload
        const payload = JSON.parse(event.body);

        // Handle different event types
        const eventType = payload.type;

        switch (eventType) {
            case 'payment.success': {
                // Handle successful payment
                const orderId = payload.orderId;
                console.log(`Processing successful payment for order
 ${orderId}`);
```

```
                // Add your business logic here
                // For example, update database, send notifications, etc.
                break;
            }

            case 'payment.failed': {
                // Handle failed payment
                const orderId = payload.orderId;
                console.log(`Processing failed payment for order ${orderId}`);

                // Add your business logic here
                break;
            }

            default:
                console.log(`Received unhandled event type: ${eventType}`);
        }

        // Return success response
        return {
            statusCode: 200,
            body: JSON.stringify({ received: true })
        };

    } catch (error) {
        if (error instanceof SyntaxError) {
            // Handle JSON parsing errors
            return {
                statusCode: 400,
                body: JSON.stringify({ error: 'Invalid JSON payload' })
            };
        }

        // Handle all other errors
        console.error('Error processing webhook:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({ error: 'Internal server error' })
        };
    }
};

// Verify the webhook signature using HMAC
```

```
const verifySignature = (event, webhookSecret) => {
    try {
        // Get the signature from headers
        const signature = event.headers['x-webhook-signature'];

        if (!signature) {
            console.log('No signature found in headers:', event.headers);
            return false;
        }

        // Get the raw body (return an empty string if the body key doesn't
 exist)
        const body = event.body || '';

        // Create HMAC using the secret key
        const hmac = crypto.createHmac('sha256', webhookSecret);
        const expectedSignature = hmac.update(body).digest('hex');

        // Compare expected and received signatures
        const isValid = signature === expectedSignature;
        if (!isValid) {
            console.log(`Invalid signature. Received: ${signature}, Expected:
 ${expectedSignature}`);
            return false;
        }

        return true;
    } catch (error) {
        console.error('Error during signature verification:', error);
        return false;
    }
 };
```

4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren.

Erstellen des geheimen Schlüssels

Damit die Lambda-Event-Funktion die Webhook-Anfrage authentifiziert, verwendet sie einen geheimen Schlüssel, den sie mit der aufrufenden Anwendung teilt. In diesem Beispiel wird der Schlüssel in einer Umgebungsvariablen gespeichert. Schließen Sie in einer Produktionsanwendung

keine sensiblen Informationen wie Passwörter in Ihren Funktionscode ein. Erstellen Sie stattdessen ein AWS Secrets Manager Geheimnis und verwenden Sie dann die Lambda-Erweiterung AWS Parameters and Secrets, um Ihre Anmeldeinformationen in Ihrer Lambda-Ereignisfunktion abzurufen.

Erstellen und Speichern des geheimen Webhook-Schlüssels

1. Generieren Sie eine lange, zufällige Zeichenfolge mit einem kryptografisch sicheren Zufallszahlengenerator. Sie können die folgenden Codefragmente in Python oder Node.js zum Generieren und Drucken eines 32-stelligen Geheimnisses verwenden oder Ihre eigene bevorzugte Methode verwenden.

   Python

   Example Code zum Generieren eines geheimen Schlüssels

   ```
   import secrets
   webhook_secret = secrets.token_urlsafe(32)
   print(webhook_secret)
   ```

   Node.js

   Example Code zum Generieren eines geheimen Schlüssels (ES-Modulformat)

   ```
   import crypto from 'crypto';
   let webhookSecret = crypto.randomBytes(32).toString('base64');
   console.log(webhookSecret)
   ```

2. Speichern Sie Ihre generierte Zeichenfolge als Umgebungsvariable für Ihre Funktion:

   a. Wählen Sie auf der Registerkarte Konfiguration für Ihre Funktion die Option Umgebungsvariablen aus.
   b. Wählen Sie Bearbeiten aus.
   c. Wählen Sie Umgebungsvariablen hinzufügen aus.
   d. Geben Sie für Schlüssel **`WEBHOOK_SECRET`** ein und dann für Wert den geheimen Schlüssel, den Sie im vorherigen Schritt generiert haben.
   e. Wählen Sie Speichern.

Sie werden diesen geheimen Schlüssel später im Tutorial erneut verwenden müssen, um Ihre Funktion zu testen. Notieren Sie ihn sich daher jetzt.

Erstellen des Funktions-URL-Endpunkts

Erstellen Sie einen Endpunkt für Ihren Webhook mithilfe einer Lambda-Funktions-URL. Da Sie den Authentifizierungstyp `NONE` verwenden, um einen Endpunkt mit öffentlichem Zugriff zu erstellen, kann jeder, der über die URL verfügt, Ihre Funktion aufrufen. Weitere Informationen zum Steuern des Zugriffs auf Funktions-URLs finden Sie unter the section called “Zugriffskontrolle”. Wenn Sie erweiterte Authentifizierungsoptionen für Ihren Webhook benötigen, sollten Sie die Verwendung von API Gateway in Betracht ziehen.

Erstellen des Funktions-URL-Endpunkts

1. Wählen Sie auf der Registerkarte Konfiguration für Ihre Event-Funktion die Option Funktions-URL aus.
2. Wählen Sie Funktions-URL erstellen.
3. Wählen Sie als Authentifizierungstyp die Option KEINEN aus.
4. Wählen Sie Speichern.

Der Endpunkt für die Funktions-URL, die Sie gerade erstellt haben, wird im Bereich Funktions-URL angezeigt. Kopieren Sie den Endpunkt, um ihn später im Tutorial zu verwenden.

Testen Sie die Event-Funktion in der Konsole

Bevor Sie eine HTTP-Anfrage verwenden, um Ihre Event-Funktion mithilfe des URL-Endpunkts aufzurufen, testen Sie sie in der Konsole, um sicherzustellen, dass Ihr Code wie erwartet funktioniert.

Um die Event-Funktion in der Konsole zu verifizieren, berechnen Sie zunächst eine Webhook-Signatur anhand des Geheimnisses, das Sie zuvor im Tutorial generiert haben, mit der folgenden JSON-Test-Payload:

```
{
    "type": "payment.success",
    "orderId": "1234",
    "amount": "99.99"
}
```

Verwenden Sie eines der folgenden Python- oder Node.js Codebeispiele, um die Webhook-Signatur mithilfe Ihres eigenen Geheimnisses zu berechnen.

Python

Berechnen der Webhook-Signatur

1. Speichern Sie den folgenden Code als Datei mit dem Namen `calculate_signature.py`. Ersetzen Sie den geheimen Webhook-Schlüssel im Code durch Ihren eigenen Wert.

```
import secrets
import hmac
import json
import hashlib

webhook_secret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"

body = json.dumps({"type": "payment.success", "orderId": "1234", "amount":
 "99.99"})

signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

print(signature)
```

2. Berechnen Sie die Signatur, indem Sie den folgenden Befehl aus dem Verzeichnis ausführen, in dem Sie den Code gespeichert haben. Kopieren Sie die Signatur, die der Code ausgibt.

```
python calculate_signature.py
```

Node.js

Berechnen der Webhook-Signatur

1. Speichern Sie den folgenden Code als Datei mit dem Namen `calculate_signature.mjs`. Ersetzen Sie den geheimen Webhook-Schlüssel im Code durch Ihren eigenen Wert.

```
import crypto from 'crypto';

const webhookSecret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"
```

```
const body = "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}";

let hmac = crypto.createHmac('sha256', webhookSecret);
let signature = hmac.update(body).digest('hex');

console.log(signature);
```

2. Berechnen Sie die Signatur, indem Sie den folgenden Befehl aus dem Verzeichnis ausführen, in dem Sie den Code gespeichert haben. Kopieren Sie die Signatur, die der Code ausgibt.

```
node calculate_signature.mjs
```

Sie können Ihren Funktionscode jetzt mit einer Test-HTTP-Anfrage in der Konsole testen.

Testen Sie die Event-Funktion in der Konsole

1. Wählen Sie die Registerkarte Code für Ihre Event-Funktion aus.
2. Wählen Sie im Abschnitt TESTEREIGNISSE die Option Neues Testereignis erstellen aus.
3. Geben Sie als Event name (Ereignisname) die Zeichenfolge **`myEvent`** ein.
4. Ersetzen Sie das vorhandene JSON, indem Sie Folgendes kopieren und in den Bereich Ereignis-JSON einfügen. Ersetzen Sie die Webhook-Signatur durch den im vorherigen Schritt berechneten Wert.

```
{
  "headers": {
    "Content-Type": "application/json",
    "x-webhook-signature":
 "2d672e7a0423fab740fbc040e801d1241f2df32d2ffd8989617a599486553e2a"
  },
  "body": "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}"
}
```

5. Wählen Sie Speichern.
6. Wählen Sie aufrufen aus.

   Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

Python

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\": true}"
}

Function Logs:
START RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Version: $LATEST
Processing successful payment for order 1234
END RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6
REPORT RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Duration: 1.55 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 36 MB Init Duration: 136.32
 ms
```

Node.js

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\":true}"
}

Function Logs:
START RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Version: $LATEST
2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO Processing
 successful payment for order 1234
END RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
REPORT RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Duration: 60.10 ms Billed
 Duration: 61 ms Memory Size: 128 MB Max Memory Used: 72 MB Init Duration:
 174.46 ms

Request ID: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
```

Testen Sie die Event-Funktion mithilfe einer HTTP-Anfrage

Testen Sie Ihren Webhook-Endpunkt mit dem Befehlszeilentool curl.

Testen Sie die Event-Funktion mithilfe von HTTP-Anfragen

1. Führen Sie in einem Terminal- oder Shell-Programm den folgenden curl-Befehl aus. Ersetzen Sie die URL durch den Wert für Ihren eigenen Funktions-URL-Endpunkt und ersetzen Sie die Webhook-Signatur durch die Signatur, die Sie mit Ihrem eigenen geheimen Schlüssel berechnet haben.

   ```
   curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
    \
   -H "Content-Type: application/json" \
   -H "x-webhook-
   signature: d5f52b76ffba65ff60ea73da67bdf1fc5825d4db56b5d3ffa0b64b7cb85ef48b" \
   -d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
   ```

   Die Ausgabe sollte folgendermaßen aussehen:

   ```
   {"received": true}
   ```

2. Untersuchen Sie die CloudWatch Protokolle Ihrer Funktion, um sicherzustellen, dass sie die Nutzlast korrekt analysiert hat. Gehen Sie dazu wie folgt vor:

   a. Öffnen Sie die Gruppenseite Logs in der CloudWatch Amazon-Konsole.

   b. Wählen Sie die Protokollgruppe Ihrer Funktion (`/aws/lambda/myLambdaWebhook`).

   c. Wählen Sie den neuesten Protokoll-Stream.

      In den Protokollen Ihrer Funktion sollte eine Ausgabe ähnlich der folgenden zu sehen sein:

      Python

      ```
      Processing successful payment for order 1234
      ```

      Node.js

      ```
      2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO
       Processing successful payment for order 1234
      ```

3. Überprüfen Sie, ob Ihr Code eine ungültige Signatur erkennt, indem Sie den folgenden curl-Befehl ausführen. Ersetzen Sie die URL durch Ihren eigenen Funktions-URL-Endpunkt.

```
curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
 \
-H "Content-Type: application/json" \
-H "x-webhook-signature: abcdefg" \
-d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
```

   Die Ausgabe sollte folgendermaßen aussehen:

```
{"error": "Invalid signature"}
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Als Sie die Lambda-Ereignisfunktion in der Konsole erstellt haben, hat Lambda auch eine Ausführungsrolle für Ihre Funktion erstellt.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie die von Lambda erstellte Ausführungsrolle aus. Die Rolle weist das Benennungsformat `myLambdaWebhook-role-<random string>` auf.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Die Skalierung von Lambda-Ereignisfunktionen verstehen

Parallelität ist die Anzahl der laufenden Anfragen, die Ihre AWS Lambda Event-Funktion gleichzeitig bearbeitet. Für jede gleichzeitige Anfrage stellt Lambda eine separate Instance Ihrer Ausführungsumgebung bereit. Wenn Ihre Event Functions mehr Anfragen erhalten, übernimmt Lambda automatisch die Skalierung der Anzahl der Ausführungsumgebungen, bis Sie das Parallelitätslimit Ihres Kontos erreicht haben. Standardmäßig bietet Lambda Ihrem Konto ein Gesamt-Parallelitätslimit von 1.000 gleichzeitigen Ausführungen für alle Event-Funktionen in einem. AWS-Region Um Ihren spezifischen Kontoanforderungen gerecht zu werden, können Sie eine Erhöhung des Kontingents beantragen und Parallelitätskontrollen auf Funktionsebene konfigurieren, sodass Ihre kritischen Event-Funktionen nicht gedrosselt werden.

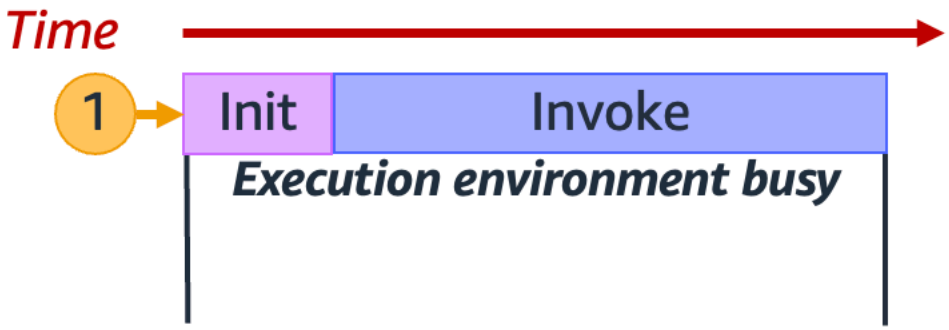
In diesem Thema werden Parallelitätskonzepte und die Skalierung von Event Functions in Lambda erklärt. Am Ende dieses Themas werden Sie in der Lage sein, die Gleichzeitigkeit zu berechnen, die beiden Hauptoptionen für die Gleichzeitigkeitskontrolle (reserviert und bereitgestellt) zu visualisieren, geeignete Einstellungen für die Gleichzeitigkeitskontrolle zu schätzen und Metriken zur weiteren Optimierung anzuzeigen.

Sections

- Verstehen und Visualisieren der Gleichzeitigkeit
- Berechnung der Parallelität für eine Event-Funktion
- Verständnis der reservierten Gleichzeitigkeit und der bereitgestellten Gleichzeitigkeit
- Gleichzeitigkeit und Anforderungen pro Sekunde verstehen
- Gleichzeitigkeitskontingente
- Konfiguration reservierter Parallelität für eine Event-Funktion
- Konfiguration der bereitgestellten Parallelität für eine Event-Funktion
- Skalierungsverhalten der Event-Funktion
- Überwachen der Gleichzeitigkeit

Verstehen und Visualisieren der Gleichzeitigkeit

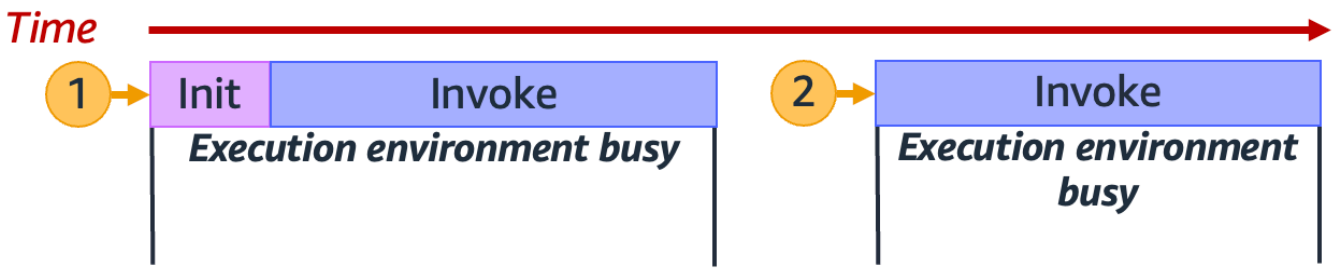
Lambda ruft Ihre Event Function in einer sicheren und isolierten Ausführungsumgebung auf. Um eine Anfrage zu bearbeiten, muss Lambda zuerst eine Ausführungsumgebung initialisieren (die Init-Phase), bevor es sie zum Aufrufen Ihrer Event-Funktion (die Aufrufphase) verwendet:



> **Note**
>
> Die tatsächliche Dauer der Initialisierung und des Aufrufs kann in Abhängigkeit von vielen Faktoren variieren, z. B. von der ausgewählten Laufzeit und dem Code der Lambda-Ereignisfunktion. Das vorherige Diagramm soll nicht die genauen Proportionen der Initiierungs- und Aufrufphasendauer darstellen.

Im vorherigen Diagramm wird ein Rechteck verwendet, um eine einzelne Ausführungsumgebung darzustellen. Wenn Ihre Event-Funktion ihre allererste Anfrage erhält (dargestellt durch den gelben Kreis mit Bezeichnung1), erstellt Lambda eine neue Ausführungsumgebung und führt den Code während der Init-Phase außerhalb Ihres Haupthandlers aus. Dann führt Lambda den Haupthandlercode Ihrer Event-Funktion während der Aufrufphase aus. Während des gesamten Prozesses ist diese Ausführungsumgebung ausgelastet und kann keine anderen Anfragen verarbeiten.

Wenn Lambda die Verarbeitung der ersten Anfrage abgeschlossen hat, kann diese Ausführungsumgebung weitere Anfragen für dieselbe Event-Funktion verarbeiten. Für nachfolgende Anfragen muss Lambda die Umgebung nicht neu initialisieren.

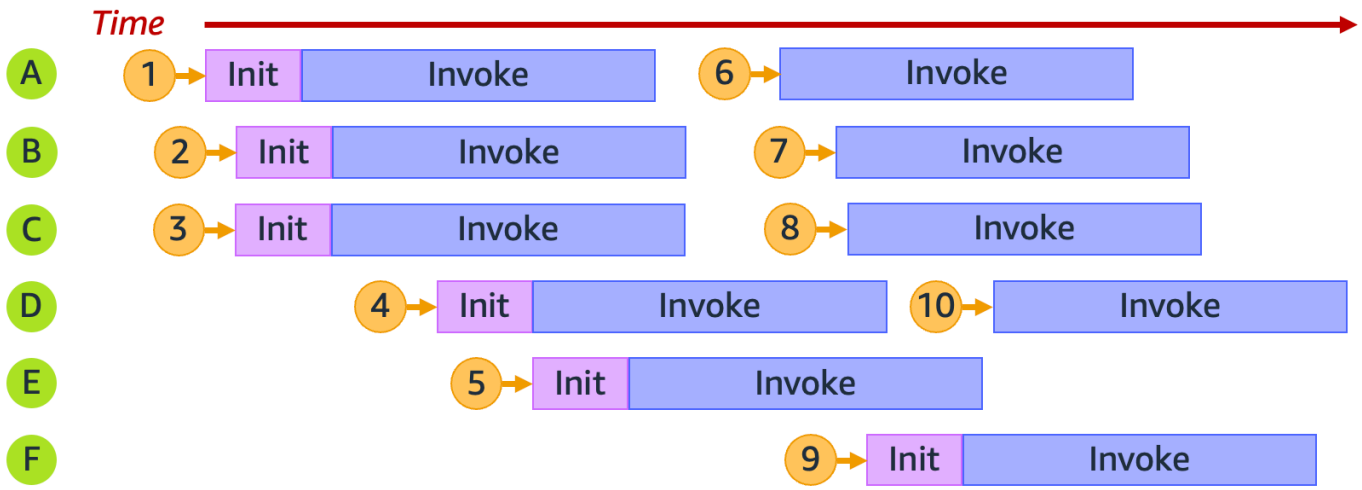


Im vorherigen Diagramm verwendet Lambda die Ausführungsumgebung erneut, um die zweite Anfrage zu verarbeiten (dargestellt durch den gelben Kreis mit der Bezeichnung 2).

Bisher haben wir uns nur auf eine einzige Instance Ihrer Ausführungsumgebung konzentriert (d. h. eine Gleichzeitigkeit von 1). In der Praxis muss Lambda möglicherweise mehrere Instanzen der Ausführungsumgebung parallel bereitstellen, um alle eingehenden Anfragen zu verarbeiten. Wenn Ihre Event-Funktion eine neue Anfrage erhält, kann eines von zwei Dingen passieren:

- Wenn eine vorinitialisierte Instance der Ausführungsumgebung verfügbar ist, verwendet Lambda diese, um die Anfrage zu verarbeiten.
- Andernfalls erstellt Lambda eine neue Instance der Ausführungsumgebung, um die Anfrage zu verarbeiten.

Lassen Sie uns zum Beispiel untersuchen, was passiert, wenn Ihre Event-Funktion 10 Anfragen erhält:



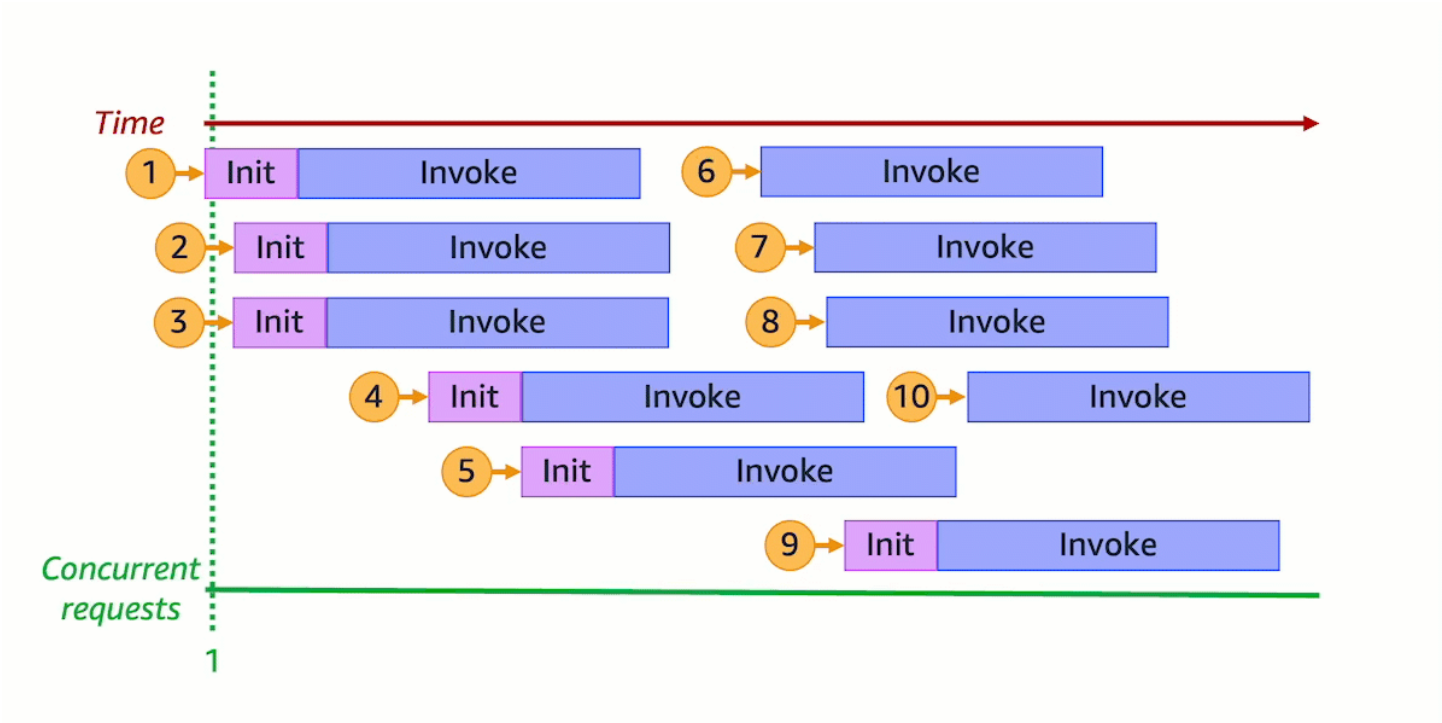
Im vorherigen Diagramm stellt jede horizontale Ebene eine einzelne Instance der Ausführungsumgebung dar (von A bis F). So behandelt Lambda jede Anfrage:

Anforderung	Lambda-Verhalten	Reasoning
1	Stellt neue Umgebung A bereit	Dies ist die erste Anfrage. Es sind keine Instances der Ausführungsumgebung verfügbar

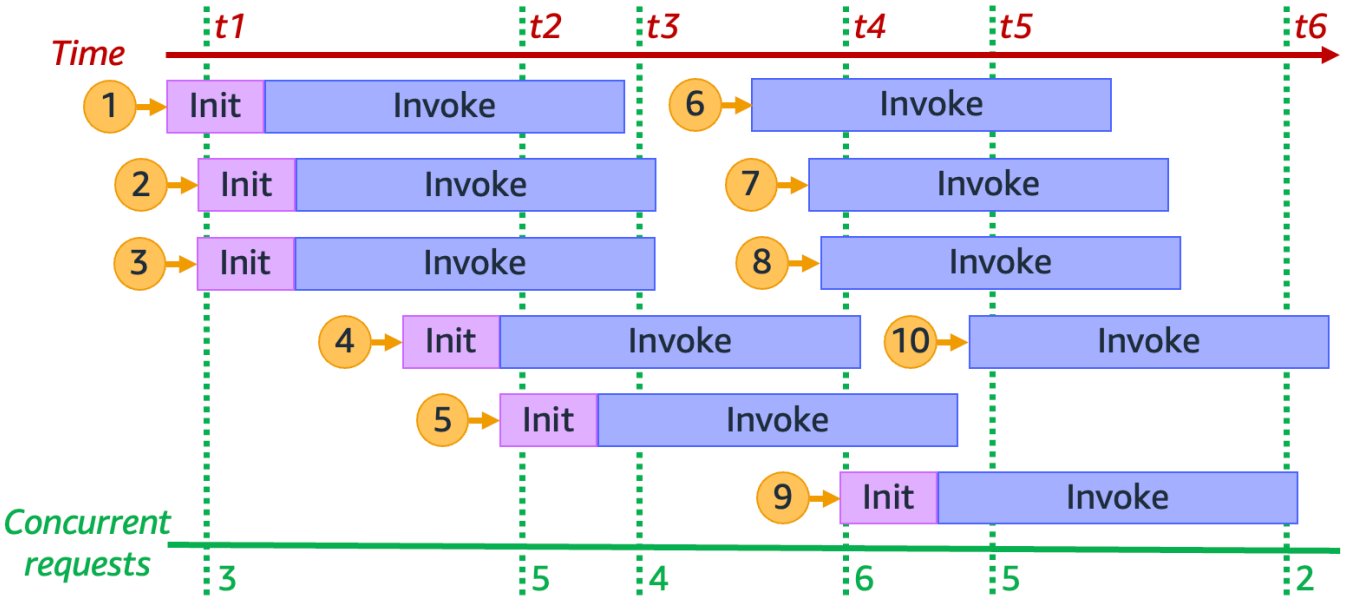
Anforderung	Lambda-Verhalten	Reasoning
2	Stellt neue Umgebung B bereit	Die vorhandene Instance der Ausführungsumgebung A ist ausgelastet
3	Stellt neue Umgebung C bereit	Die vorhandenen Instances der Ausführungsumgebung A und B sind beide ausgelastet
4	Stellt neue Umgebung D bereit	Die vorhandenen Instances der Ausführungsumgebung A, B und C sind alle ausgelastet
5	Stellt neue Umgebung E bereit	Die vorhandenen Instances der Ausführungsumgebung A, B, C und D sind alle ausgelast et
6	Wiederverwendung von Umgebung A	Die Instance der Ausführun gsumgebung A hat die Verarbeitung von Anfrage 1 abgeschlossen und ist jetzt verfügbar
7	Wiederverwendung von Umgebung B	Die Instance der Ausführun gsumgebung B hat die Verarbeitung von Anfrage 2 abgeschlossen und ist jetzt verfügbar
8	Wiederverwendung von Umgebung C	Die Instance der Ausführun gsumgebungB hat die Verarbeitung von Anfrage 3 abgeschlossen und ist jetzt verfügbar

Anforderung	Lambda-Verhalten	Reasoning
9	Stellt neue Umgebung F bereit	Die vorhandenen Instances der Ausführungsumgebung A, B, C, D und E sind alle ausgelastet
10	Wiederverwendung von Umgebung D	Die Instance der Ausführungsumgebung D hat die Verarbeitung von Anfrage 4 abgeschlossen und ist jetzt verfügbar

Wenn Ihre Event-Funktion mehr gleichzeitige Anfragen empfängt, erhöht Lambda als Reaktion darauf die Anzahl der Instanzen der Ausführungsumgebung. Die folgende Animation verfolgt die Anzahl der gleichzeitigen Anfragen im Laufe der Zeit:



Durch das Einfrieren der vorherigen Animation an sechs verschiedenen Zeitpunkten erhalten wir das folgende Diagramm:



Im vorherigen Diagramm können wir zu jedem Zeitpunkt eine vertikale Linie zeichnen und die Anzahl der Umgebungen zählen, die diese Linie schneiden. Dadurch erhalten wir die Anzahl der gleichzeitigen Anfragen zu diesem Zeitpunkt. Beispielsweise gibt es zum Zeitpunkt `t1` drei aktive Umgebungen, die drei gleichzeitige Anfragen bearbeiten. Die maximale Anzahl gleichzeitiger Anfragen in dieser Simulation wird zum Zeitpunkt `t4` erreicht, an dem sechs aktive Umgebungen sechs gleichzeitige Anfragen bearbeiten.

Zusammenfassend lässt sich sagen, dass die Parallelität Ihrer Event-Funktion die Anzahl der gleichzeitigen Anfragen ist, die sie gleichzeitig bearbeitet. Als Reaktion auf eine Zunahme der Parallelität Ihrer Event-Funktion stellt Lambda mehr Instanzen der Ausführungsumgebung bereit, um den Anforderungsbedarf zu decken.

Berechnung der Parallelität für eine Event-Funktion

Im Allgemeinen ist die Gleichzeitigkeit eines Systems die Fähigkeit, mehr als eine Aufgabe gleichzeitig zu bearbeiten. In Lambda ist Parallelität die Anzahl der laufenden Anfragen, die Ihre Event-Funktion gleichzeitig bearbeitet. Eine schnelle und praktische Methode zur Messung der Parallelität einer Lambda-Ereignisfunktion ist die Verwendung der folgenden Formel:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Gleichzeitigkeit unterscheidet sich von Anfragen pro Sekunde. Nehmen wir zum Beispiel an, Ihre Event-Funktion empfängt im Durchschnitt 100 Anfragen pro Sekunde. Wenn die durchschnittliche Dauer einer Anfrage 1 Sekunde beträgt, beträgt die Nebenläufigkeit ebenfalls 100:

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

Wenn die durchschnittliche Anfragedauer jedoch 500 ms beträgt, liegt die Nebenläufigkeit bei 50:

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

Was bedeutet eine Gleichzeitigkeit von 50 in der Praxis? Wenn die durchschnittliche Anfragedauer 500 ms beträgt, können Sie davon ausgehen, dass eine Instance Ihrer Funktion zwei Anfragen pro Sekunde verarbeiten kann. Dementsprechend sind 50 Instances Ihrer Funktion erforderlich, um eine Last von 100 Anfragen pro Sekunde zu verarbeiten. Eine Gleichzeitigkeit von 50 bedeutet, dass Lambda 50 Instances der Ausführungsumgebung bereitstellen muss, um diesen Workload ohne Drosselung effizient zu bewältigen. So drückt man dies in Gleichungsform aus:

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

Wenn Ihre Ereignisfunktion die doppelte Anzahl von Anfragen empfängt (200 Anfragen pro Sekunde), aber nur die Hälfte der Zeit benötigt, um jede Anfrage zu verarbeiten (250 ms), dann ist die Parallelität immer noch 50:

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

Testen Sie Ihr Verständnis von Gleichzeitigkeit

Angenommen, Sie haben eine Ereignisfunktion, deren Ausführung im Durchschnitt 200 ms benötigt. Während der Spitzenlast beobachten Sie 5 000 Anfragen pro Sekunde. Wie hoch ist die Parallelität Ihrer Event-Funktion bei Spitzenlast?

Antwort

Die durchschnittliche Dauer der Event-Funktion beträgt 200 ms oder 0,2 Sekunden. Mithilfe der Nebenläufigkeitsformel können Sie die Zahlen einfügen, um eine Nebenläufigkeit von 1 000 zu erhalten:

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

Alternativ bedeutet eine durchschnittliche Dauer von 200 ms, dass Ihre Event-Funktion 5 Anfragen pro Sekunde verarbeiten kann. Um den Workload von 5 000 Anfragen pro Sekunde zu bewältigen, benötigen Sie 1 000 Instances der Ausführungsumgebung. Die Gleichzeitigkeit beträgt also 1 000:

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

Verständnis der reservierten Gleichzeitigkeit und der bereitgestellten Gleichzeitigkeit

Standardmäßig gilt für Ihr Konto ein Parallelitätslimit von 1.000 gleichzeitigen Ausführungen für alle Event-Funktionen in einer Region. Ihre Event Functions teilen sich diesen Pool von 1.000 gleichzeitigen Vorgängen auf On-Demand-Basis. Bei Ihren Event-Funktionen kommt es zu einer Drosselung (d. h. sie fangen an, Anfragen zu verwerfen), wenn Ihnen die verfügbare Parallelität ausgeht.

Einige Ihrer Event-Funktionen sind möglicherweise wichtiger als andere. Aus diesem Grund möchten Sie möglicherweise Parallelitätseinstellungen konfigurieren, um sicherzustellen, dass wichtige Ereignisfunktionen die Parallelität erhalten, die sie benötigen. Es gibt zwei Arten von Gleichzeitigkeitskontrollen: reservierte Gleichzeitigkeit und bereitgestellte Gleichzeitigkeit.

- Verwenden Sie reservierte Parallelität, um sowohl die maximale als auch die minimale Anzahl gleichzeitiger Instances festzulegen, um einen Teil der Parallelität Ihres Kontos für eine Event-Funktion zu reservieren. Dies ist nützlich, wenn Sie nicht möchten, dass andere Event-Funktionen die gesamte verfügbare, nicht reservierte Parallelität in Anspruch nehmen. Wenn für eine Ereignisfunktion Parallelität reserviert ist, kann keine andere Ereignisfunktion diese Parallelität verwenden.
- Verwenden Sie die bereitgestellte Parallelität, um eine Reihe von Umgebungsinstanzen für eine Ereignisfunktion vorab zu initialisieren. Dies ist nützlich, um Kaltstartlatenzen zu reduzieren.

Reservierte Gleichzeitigkeit

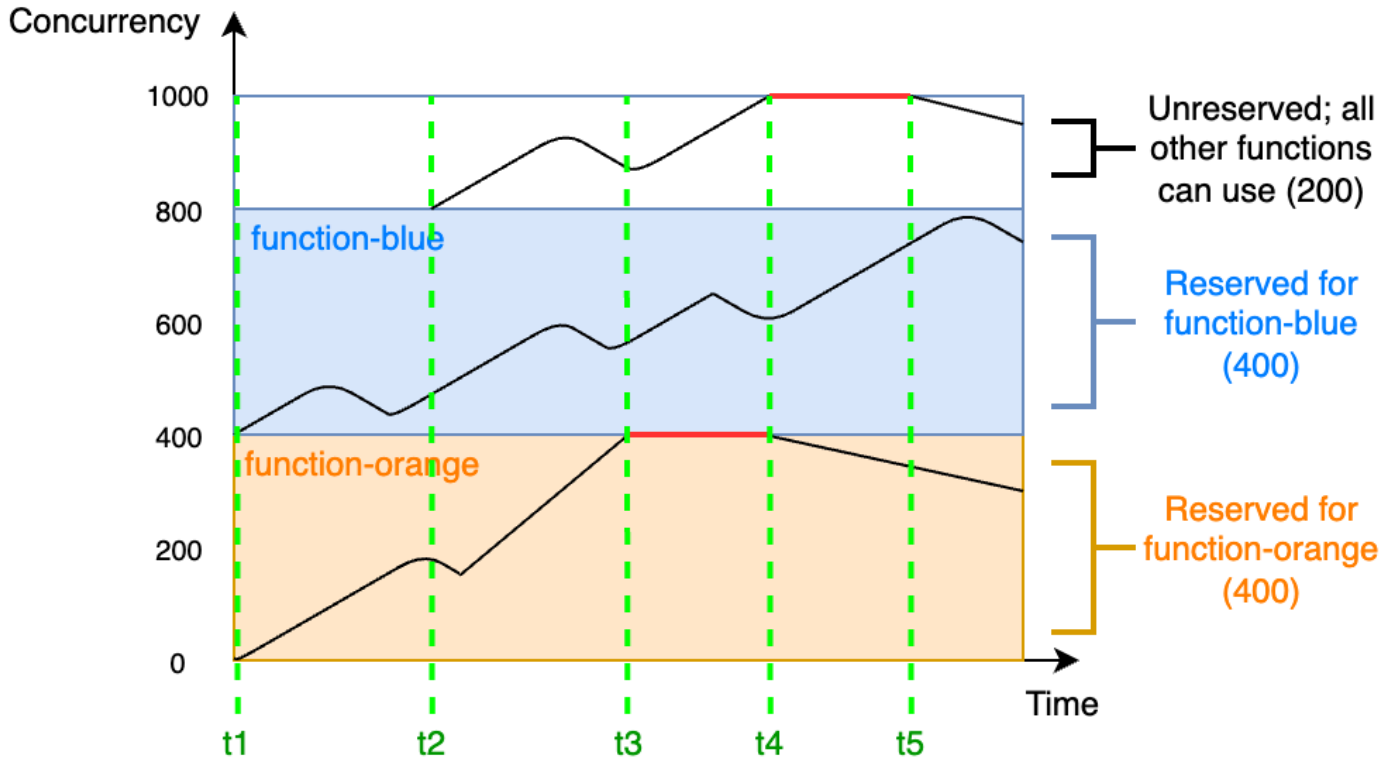
Wenn Sie sicherstellen möchten, dass für Ihre Event-Funktion jederzeit ein gewisses Maß an Parallelität verfügbar ist, verwenden Sie reservierte Parallelität.

Reservierte Parallelität legt die maximale und minimale Anzahl gleichzeitiger Instances fest, die Sie Ihrer Event-Funktion zuordnen möchten. Wenn Sie einer Event-Funktion reservierte Parallelität zuweisen, kann keine andere Event-Funktion diese Parallelität verwenden. Mit anderen Worten, die

Einstellung einer reservierten Parallelität kann sich auf den Parallelitätspool auswirken, der anderen Ereignisfunktionen zur Verfügung steht. Eventfunktionen, die nicht über reservierte Parallelität verfügen, teilen sich den verbleibenden Pool an nicht reservierter Parallelität.

Das Konfigurieren der reservierten Gleichzeitigkeit wird auf das gesamte Gleichzeitigkeitslimit Ihres Kontos angerechnet. Für die Konfiguration der reservierten Parallelität für eine Event-Funktion fallen keine Gebühren an.

Betrachten Sie das folgende Diagramm, um die reservierte Gleichzeitigkeit besser zu verstehen:



In diesem Diagramm liegt das Limit für die Parallelität Ihres Kontos für alle Event-Funktionen in dieser Region auf dem Standardlimit von 1.000. Angenommen, Sie haben zwei kritische Ereignisfunktionen`function-orange`, `function-blue` die routinemäßig mit hohen Aufrufvolumen rechnen. Sie entscheiden sich, 400 Einheiten reservierter Gleichzeitigkeit für `function-blue` und 400 Einheiten reservierter Gleichzeitigkeit für `function-orange` einzusetzen. In diesem Beispiel müssen sich alle anderen Event-Funktionen in Ihrem Konto die verbleibenden 200 Einheiten der uneingeschränkten Parallelität teilen.

Das Diagramm enthält fünf wichtige Punkte:

- Bei `t1` beginnen sowohl `function-orange` als auch `function-blue` mit dem Empfang von Anfragen. Jede Event-Funktion beginnt, den ihr zugewiesenen Teil der reservierten Parallelitätseinheiten aufzubrauchen.
- Bei `t2` erhalten `function-orange` und `function-blue` stetig mehr Anfragen. Gleichzeitig stellen Sie einige andere Lambda-Ereignisfunktionen bereit, die mit dem Empfang von Anfragen beginnen. Diesen anderen Eventfunktionen weisen Sie keine reservierte Parallelität zu. Diese beginnen mit der Verwendung der verbleibenden 200 Einheiten der nicht reservierten Gleichzeitigkeit.
- Bei `t3` erreicht `function-orange` die maximale Gleichzeitigkeit von 400. Obwohl es an anderer Stelle in Ihrem Konto ungenutzte Gleichzeitigkeit gibt, kann `function-orange` nicht darauf zugreifen. Die rote Linie weist darauf hin, dass `function-orange` es zu einer Drosselung kommt und Lambda möglicherweise Anfragen verwirft.
- Bei `t4` beginnt `function-orange` weniger Anfragen zu erhalten und wird nicht mehr gedrosselt. Bei Ihren anderen Event-Funktionen kommt es jedoch zu einem Anstieg des Datenverkehrs und es kommt zu Drosselungen. Obwohl es an anderer Stelle in Ihrem Konto ungenutzte Parallelität gibt, können diese anderen Event-Funktionen nicht darauf zugreifen. Die rote Linie weist darauf hin, dass bei Ihren anderen Event-Funktionen eine Drosselung auftritt.
- Ab `t5` erhalten andere Event-Funktionen weniger Anfragen und es erfolgt keine Drosselung mehr.

Anhand dieses Beispiels können Sie sehen, dass die Reservierung von Gleichzeitigkeit folgende Auswirkungen hat:

- Ihre Event-Funktion kann unabhängig von anderen Event-Funktionen in Ihrem Konto skaliert werden. Alle Event-Funktionen Ihres Kontos in derselben Region, für die es keine reservierte Parallelität gibt, teilen sich den Pool der unreservierten Parallelität. Ohne reservierte Parallelität können andere Event-Funktionen möglicherweise Ihre gesamte verfügbare Parallelität verbrauchen. Dadurch wird verhindert, dass kritische Event-Funktionen bei Bedarf hochskaliert werden.
- Ihre Event-Funktion kann nicht außer Kontrolle geraten. Die reservierte Parallelität begrenzt die maximale und minimale Parallelität Ihrer Event-Funktion. Das bedeutet, dass Ihre Event-Funktion keine Parallelität verwenden kann, die für andere Event-Funktionen reserviert ist, oder Parallelität aus dem Pool ohne Reservierung. Darüber hinaus fungiert die reservierte Parallelität sowohl als Unter- als auch als Obergrenze. Sie reserviert die angegebene Kapazität ausschließlich für Ihre Event-Funktion und verhindert gleichzeitig, dass diese über diese Grenze hinaus skaliert wird.

Sie können Parallelität reservieren, um zu verhindern, dass Ihre Event-Funktion die gesamte verfügbare Parallelität in Ihrem Konto nutzt oder nachgelagerte Ressourcen überlastet.

- Möglicherweise können Sie nicht die gesamte verfügbare Parallelität Ihres Kontos nutzen. Das Reservieren von Parallelität wird auf das Parallelitätslimit Ihres Kontos angerechnet. Das bedeutet aber auch, dass andere Event-Funktionen diesen Teil der reservierten Parallelität nicht nutzen können. Wenn Ihre Event-Funktion nicht die gesamte Parallelität nutzt, die Sie für sie reserviert haben, verschwenden Sie diese Parallelität. Dies ist kein Problem, es sei denn, andere Event-Funktionen in Ihrem Konto könnten von der verschwendeten Parallelität profitieren.

Informationen zur Verwaltung reservierter Parallelitätseinstellungen für Ihre Event-Funktionen finden Sie unter. Konfiguration reservierter Parallelität für eine Event-Funktion

Bereitgestellte Gleichzeitigkeit

Sie verwenden reservierte Parallelität, um die maximale Anzahl von Ausführungsumgebungen zu definieren, die für eine Lambda-Ereignisfunktion reserviert sind. Keine dieser Umgebungen ist jedoch vorinitialisiert. Infolgedessen können Ihre Event-Function-Aufrufe länger dauern, da Lambda zuerst die neue Umgebung initialisieren muss, bevor Sie sie zum Aufrufen Ihrer Event-Funktion verwenden kann. Wenn Lambda eine neue Umgebung initialisieren muss, um einen Aufruf auszuführen, wird dies als Kaltstart bezeichnet. Um Kaltstarts zu vermeiden, können Sie die bereitgestellte Gleichzeitigkeit verwenden.

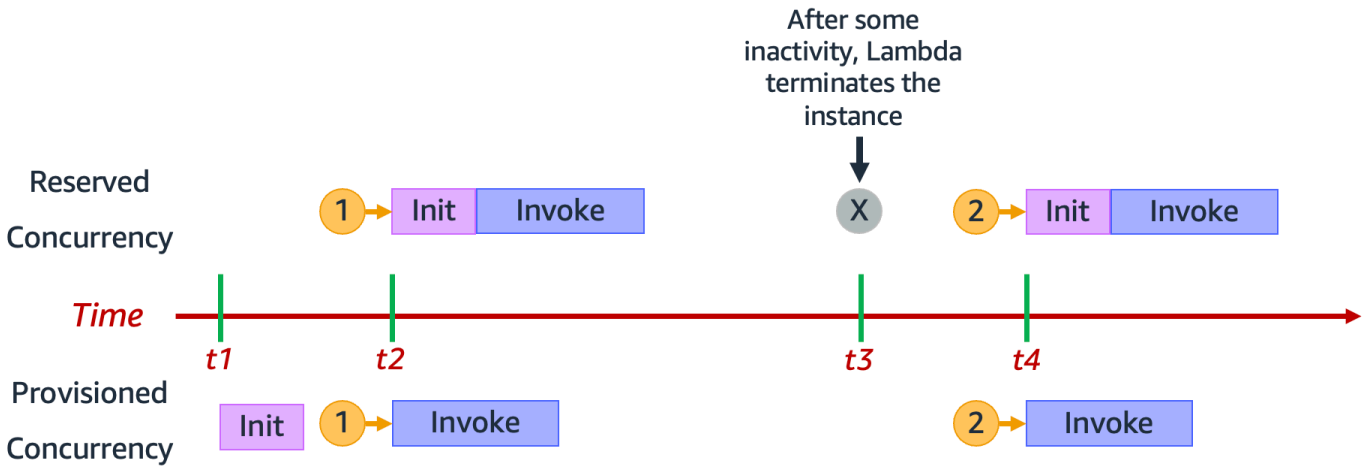
Bei der bereitgestellten Parallelität handelt es sich um die Anzahl der vorinitialisierten Ausführungsumgebungen, die Sie Ihrer Event-Funktion zuweisen möchten. Wenn Sie die bereitgestellte Parallelität für eine Event-Funktion festlegen, initialisiert Lambda diese Anzahl von Ausführungsumgebungen, sodass diese bereit sind, sofort auf Anfragen von Event-Funktionen zu reagieren.

> **Note**
>
> Die Verwendung von bereitgestellter Gleichzeitigkeit verursacht zusätzliche Gebühren für Ihr Konto. Wenn Sie mit den Laufzeiten Java 11 oder Java 17 arbeiten, können Sie Lambda auch verwenden, SnapStart um Kaltstartprobleme ohne zusätzliche Kosten zu beheben. SnapStart verwendet zwischengespeicherte Snapshots Ihrer Ausführungsumgebung, um die Startleistung erheblich zu verbessern. Sie können nicht beide SnapStart und die bereitgestellte Parallelität in derselben Version der Event Function verwenden. Weitere

Informationen zu SnapStart Funktionen, Einschränkungen und unterstützten Regionen finden Sie unter. Verbesserung der Startleistung mit Lambda SnapStart

Bei der Verwendung von bereitgestellter Gleichzeitigkeit erneuert Lambda weiterhin die Ausführungsumgebung im Hintergrund. Dies kann beispielsweise nach einem Aufruffehler der Fall sein. Lambda stellt jedoch zu jedem Zeitpunkt sicher, dass die Anzahl der vorinitialisierten Umgebungen dem Wert der bereitgestellten Parallelitätseinstellung Ihrer Event-Funktion entspricht. Wichtig: Selbst wenn Sie die bereitgestellte Parallelität verwenden, kann es dennoch zu einer Kaltstartverzögerung kommen, wenn Lambda die Ausführungsumgebung zurücksetzen muss.

Im Gegensatz dazu kann Lambda bei Verwendung von reservierter Parallelität eine Umgebung nach einer gewissen Zeit der Inaktivität vollständig beenden. Das folgende Diagramm veranschaulicht dies, indem der Lebenszyklus einer einzelnen Ausführungsumgebung verglichen wird, wenn Sie Ihre Event Function mit reservierter Parallelität und bereitgestellter Parallelität konfigurieren.

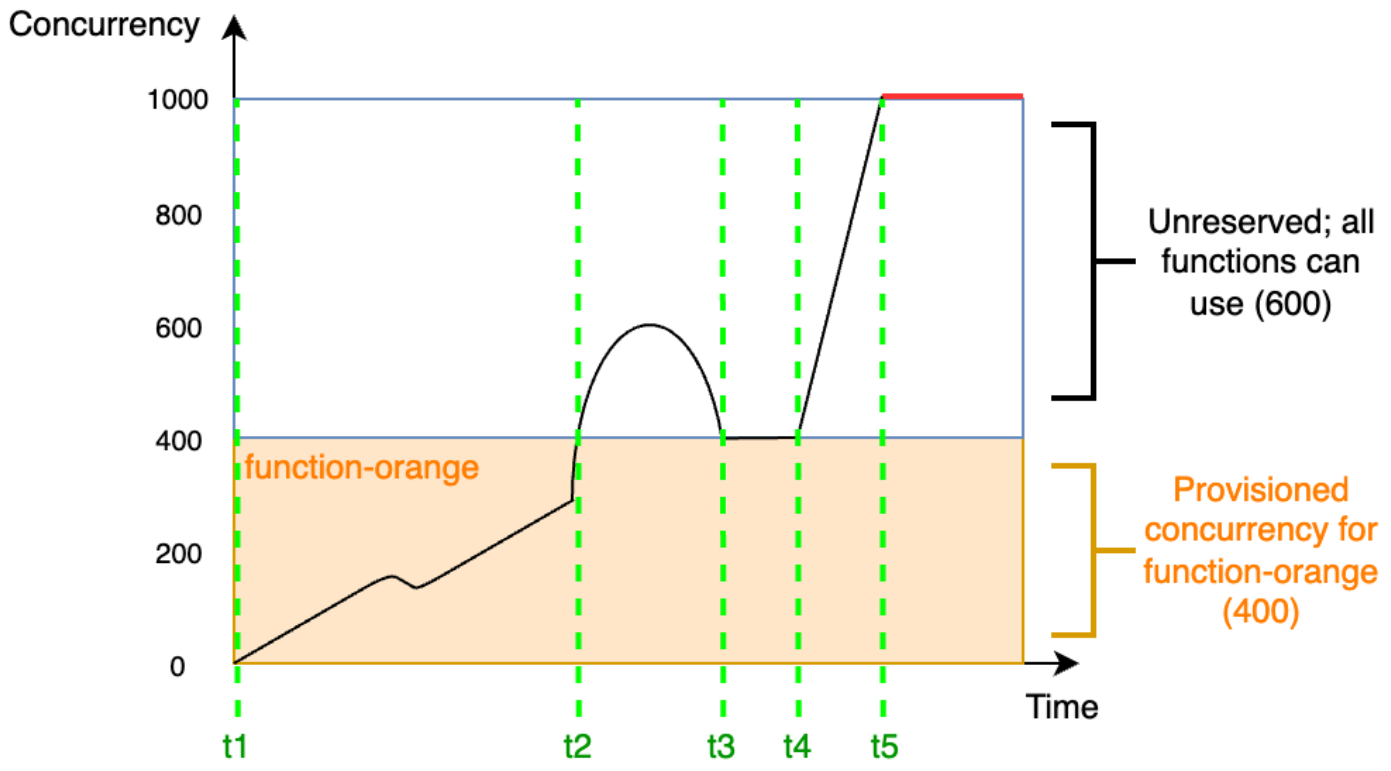


Das Diagramm enthält vier wichtige Punkte:

Zeit	Reservierte Gleichzeitigkeit	Bereitgestellte Gleichzeitigkeit
t1	Es passiert nichts.	Lambda initialisiert eine Instance der Ausführun gsumgebung vorab.
t2	Anfrage 1 wird empfangen . Lambda muss eine neue	Anfrage 1 wird empfangen . Lambda verwendet die

Zeit	Reservierte Gleichzeitigkeit	Bereitgestellte Gleichzeitigkeit
	Instance der Ausführun gsumgebung initialisieren.	vorinitialisierte Umgebungs-Instance.
t3	Nach einer gewissen Inaktivit ät beendet Lambda die aktive Umgebungs-Instance.	Es passiert nichts.
t4	Anfrage 2 wird empfangen . Lambda muss eine neue Instance der Ausführun gsumgebung initialisieren.	Anfrage 2 wird empfangen . Lambda verwendet die vorinitialisierte Umgebungs-Instance.

Betrachten Sie das folgende Diagramm, um die bereitgestellte Gleichzeitigkeit besser zu verstehen:



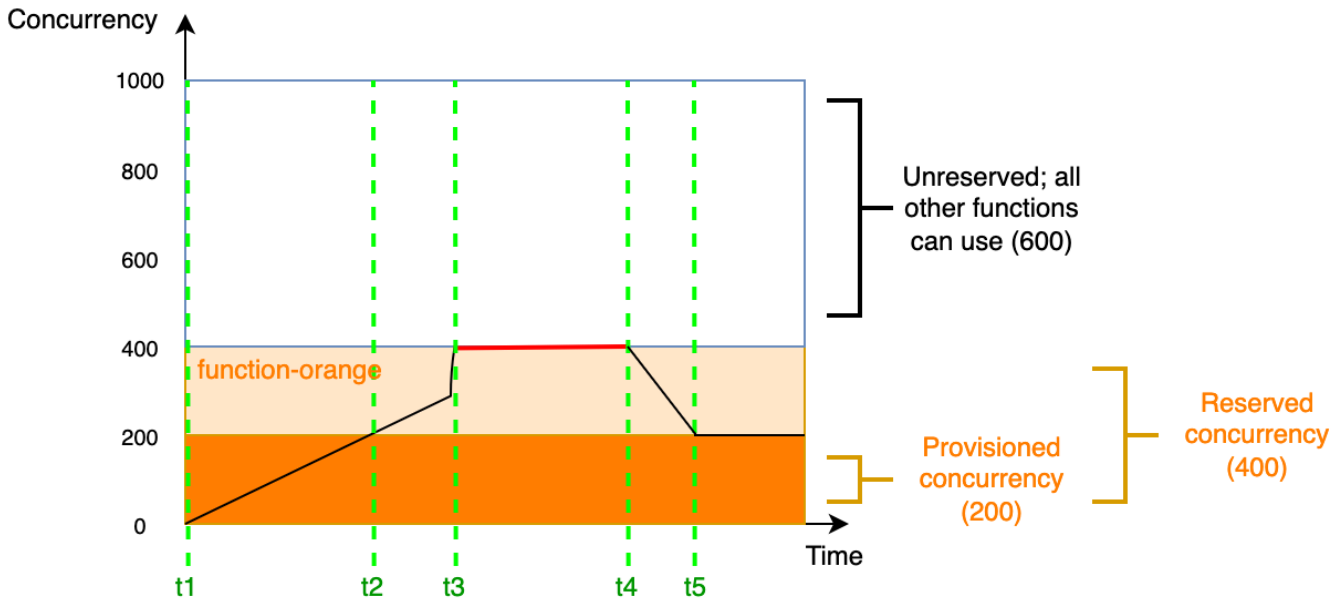
In diesem Diagramm haben Sie eine Kontogleichzeitigkeitslimit von 1 000. Sie entscheiden sich, 400 Einheiten der bereitgestellten Gleichzeitigkeit an `function-orange` zu vergeben. Alle Event-

Funktionen in Ihrem Konto, einschließlich`function-orange`, können die verbleibenden 600 Einheiten unreservierter Parallelität verwenden.

Das Diagramm enthält fünf wichtige Punkte:

- Bei `t1` beginnt `function-orange` mit dem Empfang von Anfragen. Da Lambda 400 Instances der Ausführungsumgebung vorinitialisiert hat, ist `function-orange` für den sofortigen Aufruf bereit.
- Bei `t2` erreicht `function-orange` 400 gleichzeitige Anfragen. Infolgedessen geht `function-orange` die bereitgestellte Gleichzeitigkeit aus. Da jedoch immer noch keine nicht reservierte Gleichzeitigkeit verfügbar ist, kann Lambda dies verwenden, um zusätzliche Anfragen an `function-orange` zu verarbeiten (es gibt keine Drosselung). Lambda muss neue Instanzen erstellen, um diese Anfragen zu bearbeiten, und bei Ihrer Event-Funktion kann es zu Kaltstartlatenzen kommen.
- Bei `t3` kehrt `function-orange` nach einem kurzen Anstieg des Datenverkehrs zu 400 gleichzeitigen Anfragen zurück. Lambda ist wieder in der Lage, alle Anfragen ohne Kaltstartlatenzen zu bearbeiten.
- Bei `t4` Event-Funktionen in Ihrem Konto kommt es zu einem Anstieg des Datenverkehrs. Dieser Anstieg kann von `function-orange` oder einer anderen Event-Funktion in Ihrem Konto stammen. Lambda verwendet nicht reservierte Gleichzeitigkeit, um diese Anfragen zu bearbeiten.
- Bei `t5` erreichen die Event-Funktionen in Ihrem Konto das maximale Parallelitätslimit von 1.000 und es kommt zu Drosselungen.

Im vorherigen Beispiel wurde nur die bereitgestellte Nebenläufigkeit berücksichtigt. In der Praxis können Sie für eine Event-Funktion sowohl die bereitgestellte als auch die reservierte Parallelität festlegen. Sie könnten dies tun, wenn Sie über eine Ereignisfunktion verfügen, die an Wochentagen eine konstante Anzahl von Aufrufen verarbeitet, aber am Wochenende routinemäßig Verkehrsspitzen verzeichnet. In diesem Fall könnten Sie die bereitgestellte Gleichzeitigkeit verwenden, um eine Basismenge an Umgebungen festzulegen, die Anfragen an Wochentagen verarbeiten, und die reservierte Gleichzeitigkeit verwenden, um die Spitzenlasten am Wochenende zu verarbeiten. Betrachten Sie das folgende Diagramm:



Nehmen wir in diesem Diagramm an, dass Sie 200 Einheiten bereitgestellter Gleichzeitigkeit und 400 Einheiten reservierter Gleichzeitigkeit für `function-orange` konfigurieren. Da Sie die reservierte Gleichzeitigkeit konfiguriert haben, kann `function-orange` keine der 600 Einheiten der nicht reservierten Gleichzeitigkeit verwenden.

Dieses Diagramm enthält fünf wichtige Punkte:

- Bei `t1` beginnt `function-orange` mit dem Empfang von Anfragen. Da Lambda 200 Instances der Ausführungsumgebung vorinitialisiert hat, ist `function-orange` für den sofortigen Aufruf bereit.
- Bei `function-orange` verbraucht `t2` die gesamte bereitgestellte Parallelität. `function-orange`kann weiterhin Anfragen mit reservierter Parallelität bearbeiten, aber bei diesen Anfragen kann es zu Kaltstart-Latenzen kommen.
- Bei `t3` erreicht `function-orange` 400 gleichzeitige Anfragen. Dadurch verbraucht `function-orange` die gesamte reservierte Gleichzeitigkeit. Da `function-orange` keine nicht reservierte Gleichzeitigkeit nutzen kann, werden die Anfragen gedrosselt.
- Bei `t4` beginnt `function-orange` weniger Anfragen zu erhalten und wird nicht mehr gedrosselt.
- Bei `t5` sinkt `function-orange` auf 200 gleichzeitige Anfragen, sodass alle Anfragen wieder bereitgestellte Nebenläufigkeit verwenden können (d. h. keine Kaltstartlatenzen).

Sowohl die reservierte Gleichzeitigkeit als auch die bereitgestellte Gleichzeitigkeit werden auf das Gleichzeitigkeitslimit Ihres Kontos und die regionalen Kontingente angerechnet. Mit anderen Worten, die Zuweisung von reservierter und bereitgestellter Parallelität kann sich auf den Parallelitätspool auswirken, der anderen Eventfunktionen zur Verfügung steht. Für die Konfiguration der bereitgestellten Parallelität fallen Gebühren für Sie an. AWS-Konto

> **Note**
>
> Wenn sich die Anzahl der bereitgestellten Parallelität für die Versionen und Aliase einer Event-Funktion zur reservierten Parallelität der Event-Funktion summiert, werden alle Aufrufe mit der bereitgestellten Parallelität ausgeführt. Diese Konfiguration hat auch zur Folge, dass die unveröffentlichte Version der Event-Funktion () gedrosselt wird, wodurch verhindert wird, dass sie ausgeführt wird. `$LATEST` Sie können einer Event-Funktion nicht mehr bereitgestellte Parallelität als reservierte Parallelität zuweisen.

Informationen zur Verwaltung der bereitgestellten Parallelitätseinstellungen für Ihre Eventfunktionen finden Sie unter. Konfiguration der bereitgestellten Parallelität für eine Event-Funktion Informationen zum Automatisieren der bereitgestellten Gleichzeitigkeitsskalierung basierend auf einem Zeitplan oder der Anwendungsnutzung finden Sie unter Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung.

So weist Lambda bereitgestellte Gleichzeitigkeit zu

Bereitgestellte Gleichzeitigkeit wird nicht sofort nach der Konfiguration online geschaltet. Lambda beginnt die Zuweisung bereitgestellter Gleichzeitigkeit nach ein oder zwei Minuten Vorbereitung. Für jede Event-Funktion kann Lambda jede Minute bis zu 6.000 Ausführungsumgebungen bereitstellen, unabhängig davon AWS-Region. Dies entspricht exakt der Skalierungsrate für Parallelität bei Event Functions.

Wenn Sie eine Anforderung zur Zuweisung bereitgestellter Gleichzeitigkeit einreichen, können Sie erst wieder auf diese Umgebungen zugreifen, wenn Lambda die Zuweisung vollständig abgeschlossen hat. Wenn Sie beispielsweise 5 000 bereitgestellte Gleichzeitigkeit anfordern, kann keine Ihrer Anfragen bereitgestellte Gleichzeitigkeit verwenden, bis Lambda die Zuweisung der 5 000 Ausführungsumgebungen vollständig abgeschlossen hat.

Vergleich zwischen reservierter und bereitgestellter Gleichzeitigkeit

Die folgende Tabelle fasst reservierte und bereitgestellte Nebenläufigkeit zusammen und vergleicht sie.

Thema	Reservierte Gleichzeitigkeit	Bereitgestellte Gleichzeitigkeit
Definition	Maximale Anzahl von Instanzen der Ausführun gsumgebung für Ihre Event-Funktion.	Legen Sie die Anzahl der vorab bereitgestellten Instanzen der Ausführun gsumgebung für Ihre Event-Funktion fest.
Bereitstellungsverhalten	Lambda stellt neue Instances auf On-Demand-Basis bereit.	Lambda stellt Instanzen vorab bereit (d. h. bevor Ihre Event-Funktion Anfragen empfängt).
Kaltstartverhalten	Kaltstartlatenz möglich, da Lambda bei Bedarf neue Instances erstellen muss.	Eine Kaltstartlatenz ist nicht möglich, da Lambda keine Instances On-Demand erstellen muss.
Drosselungsverhalten	Die Event-Funktion wurde gedrosselt, wenn das Limit für reservierte Parallelität erreicht wurde.	Falls reservierte Paralleli tät nicht festgelegt ist: Die Ereignisfunktion verwendet unreservierte Parallelität, wenn das bereitgestellte Parallelitätslimit erreicht ist. Falls reservierte Parallelität gesetzt ist: Die Event Function wird gedrosselt, wenn das Limit für reservierte Parallelität erreicht wurde.
Standardverhalten, falls nicht festgelegt	Event Function verwendet uneingeschränkte Parallelität,	Lambda stellt keine Instances vorab bereit. Stattdessen, falls reservierte Parallelität nicht

Thema	Reservierte Gleichzeitigkeit	Bereitgestellte Gleichzeitigkeit
	die in Ihrem Konto verfügbar ist.	festgelegt ist: Event Function verwendet uneingeschränkte Parallelität, die in Ihrem Konto verfügbar ist. Falls reservierte Parallelität gesetzt ist: Event Function verwendet reservierte Paralleli tät.
Preisgestaltung	Keine zusätzlichen Gebühren.	Verursacht zusätzliche Gebühren.

Gleichzeitigkeit und Anforderungen pro Sekunde verstehen

Wie im vorherigen Abschnitt erwähnt, unterscheidet sich die Gleichzeitigkeit von den Anforderungen pro Sekunde. Dieser Unterschied ist besonders wichtig, wenn Sie mit Event-Funktionen arbeiten, die eine durchschnittliche Anforderungsdauer von weniger als 100 ms haben.

Für alle Event-Funktionen in Ihrem Konto erzwingt Lambda ein Limit für Anfragen pro Sekunde, das dem Zehnfachen Ihrer Kontoparallelität entspricht. Da das Standardlimit für die gleichzeitige Nutzung von Konten beispielsweise bei 1.000 liegt, können die Ereignisfunktionen in Ihrem Konto maximal 10.000 Anfragen pro Sekunde verarbeiten.

Stellen Sie sich zum Beispiel eine Ereignisfunktion mit einer durchschnittlichen Anforderungsdauer von 50 ms vor. Bei 20.000 Anfragen pro Sekunde sieht die Parallelität dieser Ereignisfunktion wie folgt aus:

```
Concurrency = (20,000 requests/second) * (0.05 second/request) = 1,000
```

Ausgehend von diesem Ergebnis könnte man annehmen, dass die Gleichzeitigkeitsgrenze von 1 000 Konten ausreicht, um diese Last zu bewältigen. Aufgrund der Obergrenze von 10.000 Anfragen pro Sekunde kann Ihre Ereignisfunktion jedoch nur 10.000 Anfragen pro Sekunde von den insgesamt 20.000 Anfragen verarbeiten. Bei dieser Ereignisfunktion kommt es zu einer Drosselung.

Die Lektion besteht darin, dass Sie bei der Konfiguration der Parallelitätseinstellungen für Ihre Ereignisfunktionen sowohl Parallelität als auch Anfragen pro Sekunde berücksichtigen müssen. In

diesem Fall müssen Sie eine Erhöhung des Kontogleichzeitigkeitslimits auf 2 000 beantragen, da sich dadurch Ihr Gesamtlimit für Anfragen pro Sekunde auf 20 000 erhöhen würde.

> **Note**
>
> Basierend auf diesem Limit für Anfragen pro Sekunde ist es falsch zu sagen, dass jede Lambda-Ausführungsumgebung nur maximal zehn Anfragen pro Sekunde verarbeiten kann. Anstatt die Auslastung einer einzelnen Ausführungsumgebung zu beobachten, berücksichtigt Lambda bei der Berechnung der Quoten nur die Gesamtgleichzeitigkeit und die Gesamtanforderungen pro Sekunde.

Testen Sie Ihr Verständnis von Parallelität (Ereignisfunktionen unter 100 ms)

Angenommen, Sie haben eine Ereignisfunktion, deren Ausführung im Durchschnitt 20 ms dauert. Während der Spitzenlast beobachten Sie 30 000 Anfragen pro Sekunde. Wie hoch ist die Parallelität Ihrer Event-Funktion bei Spitzenlast?

Antwort

Die durchschnittliche Dauer der Event-Funktion beträgt 20 ms oder 0,02 Sekunden. Mithilfe der Parallelitätsformel können Sie die Zahlen einfügen, um eine Parallelität von 600 zu erhalten:

```
Concurrency = (30,000 requests/second) * (0.02 seconds/request) = 600
```

Standardmäßig scheint eine Kontogleichzeitigkeitslimit von 1 000 ausreichend zu sein, um diese Last zu bewältigen. Das Limit für Anfragen pro Sekunde von 10 000 reicht jedoch nicht aus, um die eingehenden 30 000 Anfragen pro Sekunde zu verarbeiten. Um den 30 000 Anfragen in vollem Umfang gerecht zu werden, müssen Sie eine Erhöhung des Kontogleichzeitigkeitslimits auf 3 000 oder mehr beantragen.

Das Limit für Anfragen pro Sekunde gilt für alle Kontingente in Lambda, die Gleichzeitigkeit beinhalten. Mit anderen Worten, sie gilt für synchrone On-Demand-Ereignisfunktionen, für Ereignisfunktionen, die bereitgestellte Parallelität verwenden, und für das Skalierungsverhalten bei Parallelität. Im Folgenden finden Sie beispielsweise einige Szenarien, in denen Sie sowohl Ihre Grenzwerte für Parallelität als auch für Anfragen pro Sekunde sorgfältig abwägen müssen:

- Bei einer Ereignisfunktion, die On-Demand-Parallelität verwendet, kann es zu einem Anstieg der Parallelität um 500 Mal alle 10 Sekunden oder um 5.000 Anfragen pro Sekunde alle 10 Sekunden kommen, je nachdem, was zuerst eintritt.

- Angenommen, Sie haben eine Ereignisfunktion mit einer bereitgestellten Parallelitätszuweisung von 10. Diese Ereignisfunktion überträgt sich nach 10 Parallelität oder 100 Anfragen pro Sekunde auf On-Demand-Parallelität, je nachdem, was zuerst eintritt.

Gleichzeitigkeitskontingente

Lambda legt Kontingente für die Gesamtmenge an Parallelität fest, die Sie für alle Eventfunktionen in einer Region verwenden können. Diese Kontingente bestehen auf zwei Ebenen:

- Auf Kontoebene können Ihre Event-Funktionen standardmäßig bis zu 1.000 Parallelitätseinheiten haben. Informationen zum Erhöhen dieses Limits finden Sie unter Anfordern einer Kontingenterhöhung im Benutzerhandbuch für Service Quotas.
- Auf Funktionsebene können Sie standardmäßig bis zu 900 Parallelitätseinheiten für all Ihre Event-Funktionen reservieren. Unabhängig von Ihrem Gesamtlimit für die Parallelität Ihres Kontos reserviert Lambda immer 100 Parallelitätseinheiten für Ihre Event-Funktionen, die Parallelität nicht explizit reservieren. Wenn Sie beispielsweise Ihr Limit für die Nebenläufigkeit Ihres Kontos auf 2 000 erhöht haben, können Sie auf Funktionsebene bis zu 1 900 Einheiten an Nebenläufigkeit reservieren.
- Sowohl auf Konto- als auch auf Funktionsebene erzwingt Lambda außerdem ein Limit für Anfragen pro Sekunde, das dem Zehnfachen der entsprechenden Gleichzeitigkeitsquote entspricht. Dies gilt beispielsweise für Parallelität auf Kontoebene, Eventfunktionen, die On-Demand-Parallelität verwenden, Eventfunktionen, die bereitgestellte Parallelität verwenden, und Verhalten bei der Parallelitätsskalierung. Weitere Informationen finden Sie unter the section called “Gleichzeitigkeit und Anforderungen pro Sekunde verstehen”.

Um Ihr aktuelles Kontingent für Parallelität auf Kontoebene zu überprüfen, verwenden Sie (), um den AWS Command Line Interface folgenden Befehl auszuführen:AWS CLI

```
aws lambda get-account-settings
```

Die Ausgabe sollte ungefähr wie folgt aussehen:

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
```

```
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions`ist Ihr Gesamtkontingent für Parallelität auf Kontoebene. `UnreservedConcurrentExecutions`ist die Menge an reservierter Parallelität, die Sie Ihren Event-Funktionen noch zuweisen können.

Wenn Ihre Event-Funktion mehr Anfragen erhält, skaliert Lambda automatisch die Anzahl der Ausführungsumgebungen, um diese Anfragen zu bearbeiten, bis Ihr Konto seine Parallelitätsquote erreicht hat. Zum Schutz vor einer Überskalierung als Reaktion auf plötzliche Datenverkehrsspitzen begrenzt Lambda jedoch, wie schnell Ihre Funktionen skaliert werden können. Diese Skalierungsrate für Parallelität ist die maximale Geschwindigkeit, mit der Event Functions in Ihrem Konto als Reaktion auf erhöhte Anfragen skaliert werden können. (Das heißt, sie gibt an, wie schnell Lambda neue Ausführungsumgebungen erstellen kann.) Die Skalierungsrate für Parallelität unterscheidet sich vom Parallelitätslimit auf Kontoebene, das der Gesamtmenge an Parallelität entspricht, die Ihren Eventfunktionen zur Verfügung steht.

In jeder AWS-Region und für jede Event-Funktion beträgt Ihre Parallelitätsskalierungsrate 1.000 Instanzen der Ausführungsumgebung alle 10 Sekunden (oder 10.000 Anfragen pro Sekunde alle 10 Sekunden). Mit anderen Worten: Lambda kann jeder Ihrer Funktionen alle 10 Sekunden höchstens 1 000 zusätzliche Instanzen der Ausführungsumgebung zuweisen oder 10 000 zusätzliche Anfragen pro Sekunde bearbeiten.

Normalerweise müssen Sie sich über diese Einschränkung keine Gedanken machen. Die Skalierungsrate von Lambda ist für die meisten Anwendungsfälle ausreichend.

Wichtig ist, dass die Skalierungsrate für die Gleichzeitigkeit eine Grenze auf Funktionsebene darstellt. Das bedeutet, dass jede Event-Funktion in Ihrem Konto unabhängig von anderen Event-Funktionen skaliert werden kann.

Weitere Informationen zum Skalierungsverhalten finden Sie unter Skalierungsverhalten der Event-Funktion.

Konfiguration reservierter Parallelität für eine Event-Funktion

In Lambda ist Parallelität die Anzahl der laufenden Anfragen, die Ihre Event-Funktion derzeit bearbeitet. Es gibt zwei Arten von Gleichzeitigkeitskontrollen:

- Reservierte Parallelität — Dies legt sowohl die maximale als auch die minimale Anzahl gleichzeitiger Instances fest, die Ihrer Event-Funktion zugewiesen sind. Wenn für eine Event-Funktion Parallelität reserviert ist, kann keine andere Event-Funktion diese Parallelität verwenden. Reservierte Gleichzeitigkeit ist nützlich, um sicherzustellen, dass Ihre kritischsten Funktionen immer genügend Gleichzeitigkeit haben, um eingehende Anfragen zu bearbeiten. Darüber hinaus kann reservierte Gleichzeitigkeit verwendet werden, um die Gleichzeitigkeit einzuschränken, und so zu verhindern, dass nachgelagerte Ressourcen wie Datenbankverbindungen überlastet werden. Die reservierte Gleichzeitigkeit dient sowohl als Untergrenze als auch als Obergrenze. Sie reserviert die angegebene Kapazität ausschließlich für Ihre Funktion und verhindert gleichzeitig, dass sie über diese Grenze hinaus skaliert wird. Für die Konfiguration reservierter Gleichzeitigkeit für eine Funktion wird keine zusätzliche Gebühr erhoben.
- Bereitgestellte Parallelität — Dies ist die Anzahl der vorinitialisierten Ausführungsumgebungen, die Ihrer Event-Funktion zugewiesen sind. Diese Ausführungsumgebungen sind bereit, sofort auf eingehende Funktionsanforderungen zu reagieren. Bereitgestellte Gleichzeitigkeit ist nützlich, um Kaltstartlatenzen für Funktionen zu reduzieren, und wurde entwickelt, um Funktionen mit Antwortzeiten im zweistelligen Millisekundenbereich verfügbar zu machen. Im Allgemeinen profitieren interaktive Workloads am meisten von diesem Feature. Dabei handelt es sich um Anwendungen, bei denen Benutzer Anfragen initiieren, wie z. B. Web- und Mobilanwendungen und sie reagieren am empfindlichsten auf Latenz. Asynchrone Workloads, wie Datenverarbeitungspipelines, reagieren häufig weniger empfindlich auf Latenz und benötigen daher in der Regel keine bereitgestellte Gleichzeitigkeit. Für die Konfiguration der bereitgestellten Parallelität fallen zusätzliche Gebühren für Sie an. AWS-Konto

In diesem Thema wird beschrieben, wie Sie reservierte Gleichzeitigkeit verwalten und konfigurieren. Eine konzeptionelle Übersicht über diese beiden Arten von Gleichzeitigkeitssteuerungen finden Sie unter Reservierte Gleichzeitigkeit und bereitgestellte Gleichzeitigkeit. Weitere Informationen zum Konfigurieren der bereitgestellten Gleichzeitigkeit finden Sie unter the section called “Konfigurieren von Provisioned Concurrency”.

> **Note**
>
> Lambda-Ereignisfunktionen, die mit einer Amazon MQ MQ-Ereignisquellenzuordnung verknüpft sind, haben standardmäßig eine maximale Parallelität. Für Apache Active MQ ist die maximale Anzahl der gleichzeitigen Instances 5. Für Rabbit MQ ist die maximale Anzahl der gleichzeitigen Instances 1. Wenn Sie für Ihre Event-Funktion eine reservierte oder bereitgestellte Parallelität festlegen, werden diese Grenzwerte nicht geändert. Um eine Erhöhung der standardmäßigen maximalen Parallelität bei der Verwendung von Amazon MQ zu beantragen, wenden Sie sich an. Support

Sections

- Konfigurieren reservierter Gleichzeitigkeit
- Genaue Schätzung der erforderlichen reservierten Parallelität für eine Ereignisfunktion

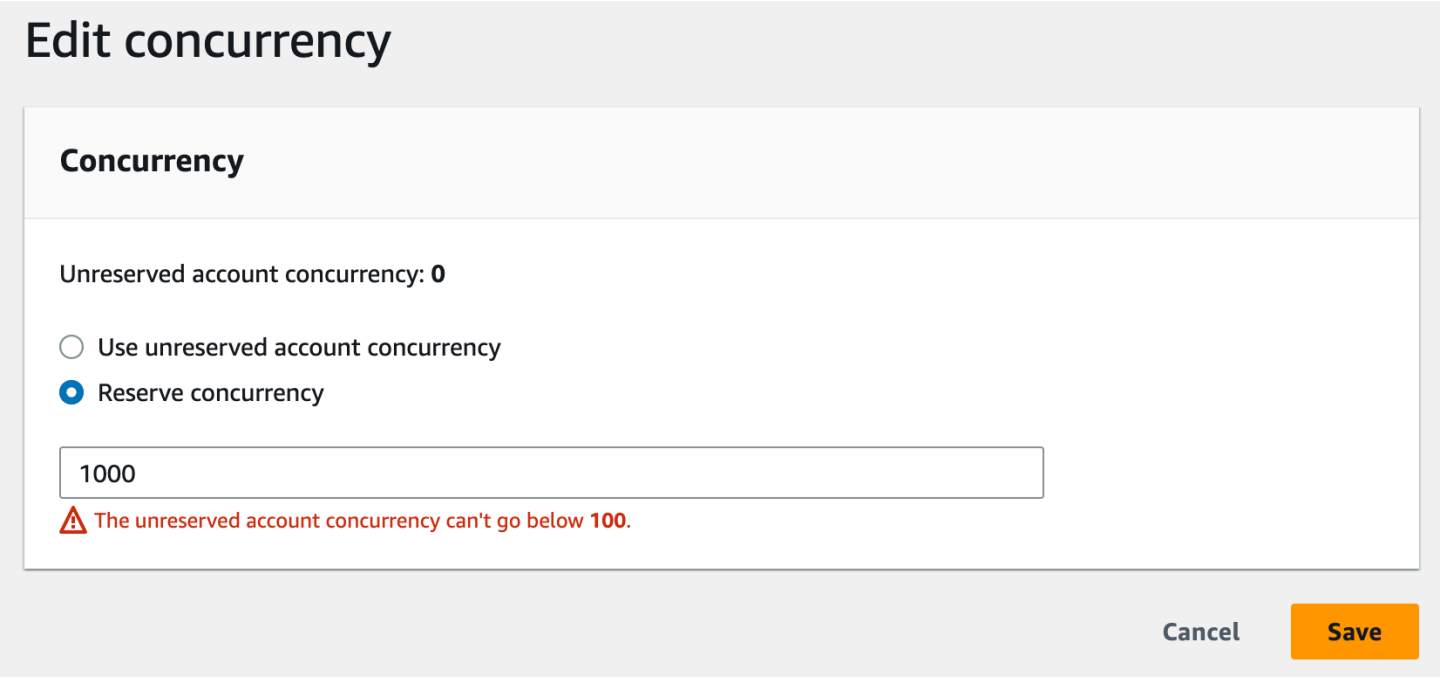
Konfigurieren reservierter Gleichzeitigkeit

Sie können reservierte Parallelitätseinstellungen für eine Lambda-Ereignisfunktion mithilfe der Lambda-Konsole oder der Lambda-API konfigurieren.

So reservieren Sie Parallelität für eine Event-Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Event-Funktion aus, für die Sie Parallelität reservieren möchten.
3. Wählen Sie Konfiguration und anschließend Gleichzeitigkeit aus.
4. Wählen Sie unter Concurrency (Parallelität) die Option Edit (Bearbeiten).
5. Wählen Sie Reserve concurrency (Parallelität reservieren). Geben Sie den Umfang der Parallelität ein, der für die Event-Funktion reserviert werden soll.
6. Wählen Sie Speichern.

Sie können bis zum Wert von Nicht reservierte Kontonebenläufigkeit minus 100 reservieren. Die verbleibenden 100 Parallelitätseinheiten sind für Event-Funktionen vorgesehen, die keine reservierte Parallelität verwenden. Wenn Ihr Konto beispielsweise ein Parallelitätslimit von 1.000 hat, können Sie nicht alle 1.000 Parallelitätseinheiten für eine einzelne Event-Funktion reservieren.



Wenn Sie Parallelität für eine Event-Funktion reservieren, wirkt sich das auf den Parallelitätspool aus, der anderen Event-Funktionen zur Verfügung steht. Wenn Sie beispielsweise 100 Parallelitätseinheiten für reservieren`function-a`, müssen andere Event-Funktionen in Ihrem Konto die verbleibenden 900 Parallelitätseinheiten gemeinsam nutzen, auch wenn `function-a` nicht alle 100 reservierten Parallelitätseinheiten verwendet werden.

Um eine Event-Funktion bewusst zu drosseln, setzen Sie ihre reservierte Parallelität auf 0. Dadurch wird verhindert, dass Ihre Event-Funktion Ereignisse verarbeitet, bis Sie das Limit aufheben.

Verwenden Sie für die Konfiguration von reservierter Gleichzeitigkeit mithilfe der Lambda-API die folgenden API-Vorgänge:

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

Um beispielsweise reservierte Parallelität mit der AWS Command Line Interface (CLI) zu konfigurieren, verwenden Sie den `put-function-concurrency` Befehl. Der folgende Befehl reserviert 100 Parallelitätseinheiten für eine Ereignisfunktion mit dem Namen: `my-function`

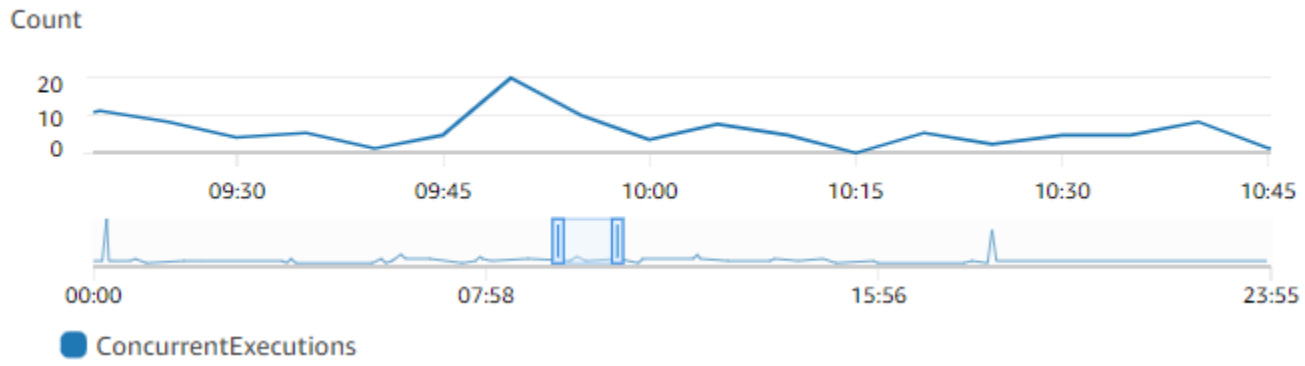
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

Die Ausgabe sollte ungefähr wie folgt aussehen:

```
{
    "ReservedConcurrentExecutions": 100
}
```

Genaue Schätzung der erforderlichen reservierten Parallelität für eine Ereignisfunktion

Wenn Ihre Event-Funktion derzeit Traffic verarbeitet, können Sie die zugehörigen Parallelitätskennzahlen einfach anhand von Metriken einsehen. CloudWatch Insbesondere zeigt Ihnen die `ConcurrentExecutions` Metrik die Anzahl der gleichzeitigen Aufrufe für jede Event-Funktion in Ihrem Konto.



Aus dem vorherigen Diagramm geht hervor, dass diese Ereignisfunktion durchschnittlich 5 bis 10 gleichzeitige Anfragen zu einem bestimmten Zeitpunkt bearbeitet und bei 20 Anfragen an einem typischen Tag ihren Höhepunkt erreicht. Nehmen wir an, Ihr Konto enthält noch viele andere Event-Funktionen. Wenn diese Ereignisfunktion für Ihre Anwendung von entscheidender Bedeutung ist und Sie keine Anfragen löschen möchten, verwenden Sie eine Zahl größer oder gleich 20 als Ihre reservierte Parallelitätseinstellung.

Denken Sie alternativ daran, dass Sie die Gleichzeitigkeit auch mit der folgenden Formel berechnen können:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Wenn Sie die durchschnittlichen Anforderungen pro Sekunde mit der durchschnittlichen Anforderungsdauer in Sekunden multiplizieren, erhalten Sie eine grobe Schätzung, wie viel Gleichzeitigkeit Sie reservieren müssen. Sie können die durchschnittlichen Anfragen pro Sekunde anhand der `Invocation`-Metrik und die durchschnittliche Anfragedauer in Sekunden anhand der

`Duration`-Metrik schätzen. Weitere Details finden Sie unter CloudWatch Metriken mit Lambda verwenden.

Sie sollten auch mit Ihren vor- und nachgelagerten Durchsatzbeschränkungen vertraut sein. Lambda Event Functions lassen sich zwar nahtlos mit der Last skalieren, aber Upstream- und Downstream-Abhängigkeiten verfügen möglicherweise nicht über die gleichen Durchsatzfähigkeiten. Wenn Sie einschränken möchten, wie hoch Ihre Event Function skalieren kann, konfigurieren Sie reservierte Parallelität für Ihre Event Function.

Konfiguration der bereitgestellten Parallelität für eine Event-Funktion

In Lambda ist Parallelität die Anzahl der In-Flight-Anfragen, die Ihre Event-Funktion derzeit bearbeitet. Es gibt zwei Arten von Gleichzeitigkeitskontrollen:

- Reservierte Parallelität — Dies legt sowohl die maximale als auch die minimale Anzahl gleichzeitiger Instances fest, die Ihrer Event-Funktion zugewiesen sind. Wenn für eine Event-Funktion Parallelität reserviert ist, kann keine andere Event-Funktion diese Parallelität verwenden. Reservierte Gleichzeitigkeit ist nützlich, um sicherzustellen, dass Ihre kritischsten Funktionen immer genügend Gleichzeitigkeit haben, um eingehende Anfragen zu bearbeiten. Darüber hinaus kann reservierte Gleichzeitigkeit verwendet werden, um die Gleichzeitigkeit einzuschränken, und so zu verhindern, dass nachgelagerte Ressourcen wie Datenbankverbindungen überlastet werden. Die reservierte Gleichzeitigkeit dient sowohl als Untergrenze als auch als Obergrenze. Sie reserviert die angegebene Kapazität ausschließlich für Ihre Funktion und verhindert gleichzeitig, dass sie über diese Grenze hinaus skaliert wird. Für die Konfiguration reservierter Gleichzeitigkeit für eine Funktion wird keine zusätzliche Gebühr erhoben.
- Bereitgestellte Parallelität — Dies ist die Anzahl der vorinitialisierten Ausführungsumgebungen, die Ihrer Event-Funktion zugewiesen sind. Diese Ausführungsumgebungen sind bereit, sofort auf eingehende Funktionsanforderungen zu reagieren. Bereitgestellte Gleichzeitigkeit ist nützlich, um Kaltstartlatenzen für Funktionen zu reduzieren, und wurde entwickelt, um Funktionen mit Antwortzeiten im zweistelligen Millisekundenbereich verfügbar zu machen. Im Allgemeinen profitieren interaktive Workloads am meisten von diesem Feature. Dabei handelt es sich um Anwendungen, bei denen Benutzer Anfragen initiieren, wie z. B. Web- und Mobilanwendungen und sie reagieren am empfindlichsten auf Latenz. Asynchrone Workloads, wie Datenverarbeitungspipelines, reagieren häufig weniger empfindlich auf Latenz und benötigen daher in der Regel keine bereitgestellte Gleichzeitigkeit. Für die Konfiguration der bereitgestellten Parallelität fallen zusätzliche Gebühren für Sie an. AWS-Konto

In diesem Thema wird beschrieben, wie Sie bereitgestellte Gleichzeitigkeit verwalten und konfigurieren. Eine konzeptionelle Übersicht über diese beiden Arten von Gleichzeitigkeitssteuerungen finden Sie unter Reservierte Gleichzeitigkeit und bereitgestellte Gleichzeitigkeit. Weitere Informationen zur Konfiguration der reservierten Gleichzeitigkeit finden Sie unter the section called “Konfigurieren reservierter Gleichzeitigkeit”.

> **Note**
>
> Lambda-Ereignisfunktionen, die mit einer Amazon MQ MQ-Ereignisquellenzuordnung verknüpft sind, haben standardmäßig eine maximale Parallelität. Für Apache Active MQ ist die maximale Anzahl der gleichzeitigen Instances 5. Für Rabbit MQ ist die maximale Anzahl der gleichzeitigen Instances 1. Wenn Sie für Ihre Event-Funktion eine reservierte oder bereitgestellte Parallelität festlegen, werden diese Grenzwerte nicht geändert. Um eine Erhöhung der standardmäßigen maximalen Parallelität bei der Verwendung von Amazon MQ zu beantragen, wenden Sie sich an. Support

Sections

- Konfigurieren von Provisioned Concurrency
- Genaue Schätzung der erforderlichen bereitgestellten Parallelität für eine Event-Funktion
- Optimieren Sie den Code der Event-Funktion bei Verwendung der bereitgestellten Parallelität
- Verwendung von Umgebungsvariablen zur Anzeige und Steuerung des bereitgestellten Gleichzeitigkeitsverhaltens
- Verstehen des Protokollierungs- und Abrechnungsverhaltens bei bereitgestellter Gleichzeitigkeit
- Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung

Konfigurieren von Provisioned Concurrency

Sie können bereitgestellte Parallelitätseinstellungen für eine Lambda-Ereignisfunktion mithilfe der Lambda-Konsole oder der Lambda-API konfigurieren.

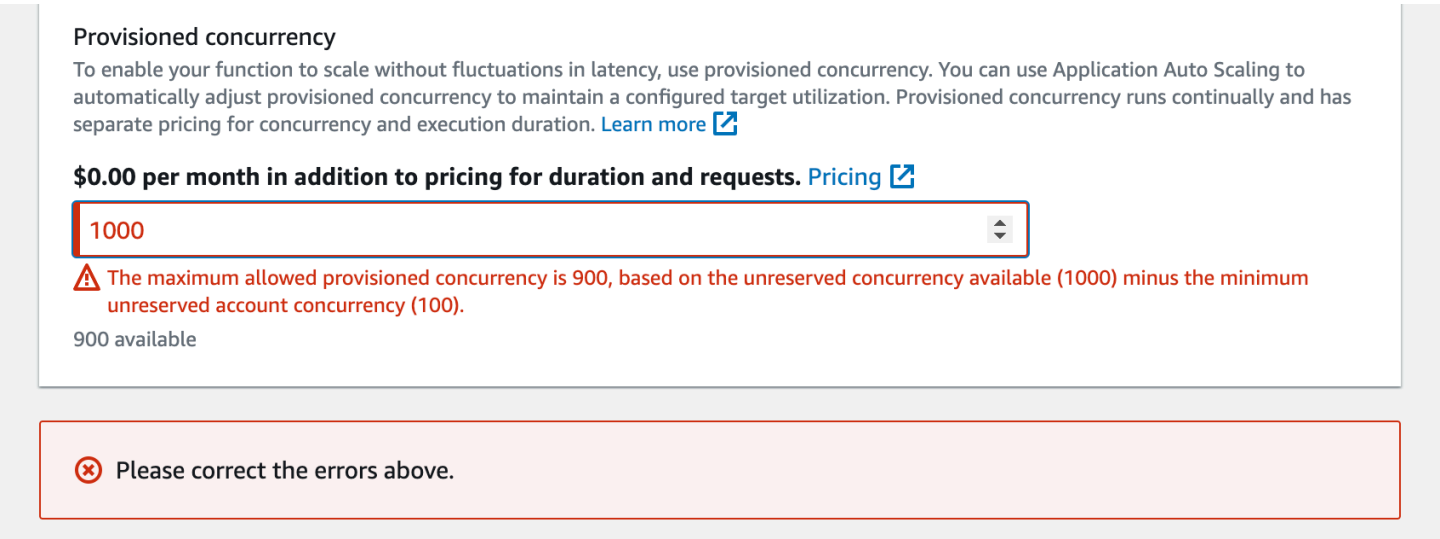
So weisen Sie einer Event-Funktion (Konsole) bereitgestellte Parallelität zu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Event-Funktion aus, der Sie bereitgestellte Parallelität zuweisen möchten.
3. Wählen Sie Konfiguration und anschließend Gleichzeitigkeit aus.
4. Wählen Sie unter Provisioned concurrency configurations (Bereitgestellte Gleichzeitigkeitskonfigurationen) die Option Add (Hinzufügen) aus.
5. Wählen Sie den Qualifizierungstyp und den Alias oder die Version aus.

> **Note**
>
> Sie können die bereitgestellte Parallelität nicht mit der $LATEST-Version einer beliebigen Event-Funktion verwenden.
> Wenn Ihre Ereignisfunktion über eine Ereignisquelle verfügt, stellen Sie sicher, dass diese Ereignisquelle auf den richtigen Alias oder die richtige Version der Ereignisfunktion verweist. Andernfalls verwendet Ihre Event-Funktion keine bereitgestellten Parallelitätsumgebungen.

6. Geben Sie unter Bereitgestellte Gleichzeitigkeit eine Zahl ein.
7. Wählen Sie Speichern.

Sie können in Ihrem Konto Gleichzeitigkeit bis zur nicht reservierten Konto-Gleichzeitigkeit minus 100 konfigurieren. Die verbleibenden 100 Parallelitätseinheiten sind für Event-Funktionen vorgesehen, die keine reservierte Parallelität verwenden. Wenn Ihr Konto beispielsweise ein Parallelitätslimit von 1.000 hat und Sie keiner Ihrer anderen Event-Funktionen eine reservierte oder bereitgestellte Parallelität zugewiesen haben, können Sie maximal 900 bereitgestellte Parallelitätseinheiten für eine einzelne Event-Funktion konfigurieren.



Die Konfiguration der bereitgestellten Parallelität für eine Event-Funktion hat Auswirkungen auf den Parallelitätspool, der anderen Eventfunktionen zur Verfügung steht. Wenn Sie beispielsweise 100 bereitgestellte Parallelitätseinheiten für konfigurieren`function-a`, müssen sich andere Eventfunktionen in Ihrem Konto die verbleibenden 900 Parallelitätseinheiten teilen. Das gilt auch, wenn `function-a` nicht alle 100 Einheiten verwendet.

Es ist möglich, derselben Event-Funktion sowohl reservierte Parallelität als auch bereitgestellte Parallelität zuzuweisen. In solchen Fällen darf die bereitgestellte Gleichzeitigkeit die reservierte Gleichzeitigkeit nicht überschreiten.

Diese Einschränkung gilt auch für Versionen von Event Function. Die maximale bereitgestellte Parallelität, die Sie einer bestimmten Version der Ereignisfunktion zuweisen können, ist die reservierte Parallelität der Ereignisfunktion abzüglich der bereitgestellten Parallelität in anderen Versionen der Ereignisfunktion.

Verwenden Sie für die Konfiguration der bereitgestellten Gleichzeitigkeit mithilfe der Lambda-API die folgenden API-Operationen:

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

Verwenden Sie beispielsweise den Befehl, um die bereitgestellte Parallelität mit der AWS Command Line Interface (CLI) zu konfigurieren. `put-provisioned-concurrency-config` Der folgende Befehl weist 100 Einheiten bereitgestellter Parallelität für den `BLUE` Alias einer Ereignisfunktion mit dem Namen zu: `my-function`

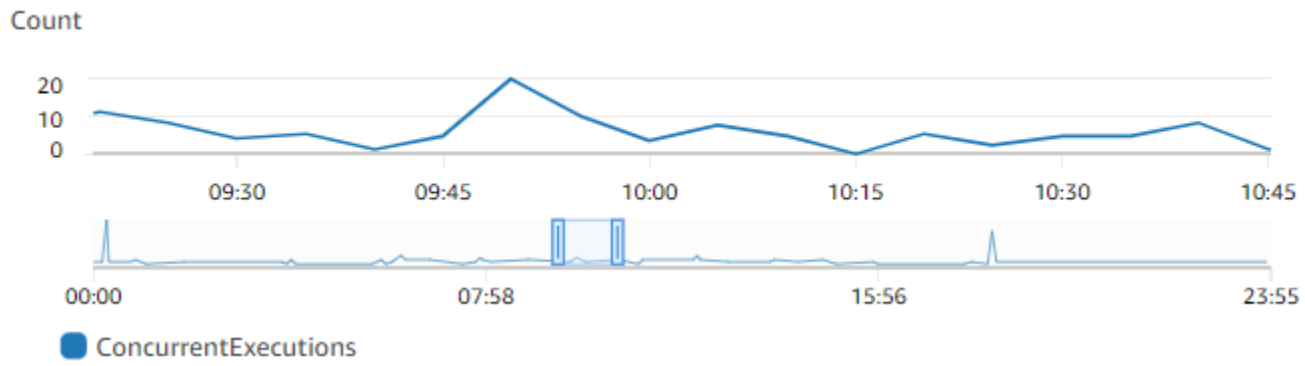
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

Die Ausgabe sollte ungefähr wie folgt aussehen:

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

Genaue Schätzung der erforderlichen bereitgestellten Parallelität für eine Event-Funktion

Sie können die Parallelitätsmetriken jeder aktiven Event-Funktion mithilfe von Metriken anzeigen. CloudWatch Insbesondere zeigt Ihnen die `ConcurrentExecutions` Metrik die Anzahl der gleichzeitigen Aufrufe von Event Functions in Ihrem Konto.



Aus dem vorherigen Diagramm geht hervor, dass diese Ereignisfunktion durchschnittlich 5 bis 10 gleichzeitige Anfragen zu einem bestimmten Zeitpunkt bearbeitet und bei 20 Anfragen ihren Höhepunkt erreicht. Nehmen wir an, Ihr Konto enthält noch viele andere Event-Funktionen. Wenn diese Ereignisfunktion für Ihre Anwendung von entscheidender Bedeutung ist und Sie bei jedem Aufruf eine Antwort mit niedriger Latenz benötigen, konfigurieren Sie mindestens 20 bereitgestellte Parallelitätseinheiten.

Denken Sie daran, dass Sie die Gleichzeitigkeit auch mit der folgenden Formel berechnen können:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Um abzuschätzen, wie viel Gleichzeitigkeit Sie benötigen, multiplizieren Sie die durchschnittlichen Anfragen pro Sekunde mit der durchschnittlichen Anfragedauer in Sekunden. Sie können die durchschnittlichen Anfragen pro Sekunde anhand der `Invocation`-Metrik und die durchschnittliche Anfragedauer in Sekunden anhand der `Duration`-Metrik schätzen.

Bei der Konfiguration der bereitgestellten Parallelität schlägt Lambda vor, zusätzlich zu der Menge an Parallelität, die Ihre Event-Funktion normalerweise benötigt, einen Puffer von 10% hinzuzufügen. Wenn Ihre Event-Funktion beispielsweise normalerweise bei 200 gleichzeitigen Anfragen ihren Höhepunkt erreicht, legen Sie die bereitgestellte Parallelität auf 220 fest (200 gleichzeitige Anfragen + 10% = 220 bereitgestellte Parallelität).

Optimieren Sie den Code der Event-Funktion bei Verwendung der bereitgestellten Parallelität

Wenn Sie bereitgestellte Parallelität verwenden, sollten Sie erwägen, Ihren Event-Funktionscode umzustrukturieren, um ihn für eine geringe Latenz zu optimieren. Für Eventfunktionen, die bereitgestellte Parallelität verwenden, führt Lambda während der Zuweisungszeit jeden Initialisierungscode aus, z. B. das Laden von Bibliotheken und das Instanziieren von Clients. Daher ist es ratsam, möglichst viele Initialisierungen außerhalb des Haupthandlers für Ereignisfunktionen zu verschieben, um zu vermeiden, dass die Latenz bei tatsächlichen Aufrufen der Ereignisfunktion beeinträchtigt wird. Im Gegensatz dazu bedeutet das Initialisieren von Bibliotheken oder das Instanziieren von Clients innerhalb Ihres Haupthandlercodes, dass Ihre Event-Funktion dies bei jedem Aufruf ausführen muss (dies geschieht unabhängig davon, ob Sie die bereitgestellte Parallelität verwenden).

Bei Aufrufen auf Abruf muss Lambda Ihren Initialisierungscode möglicherweise jedes Mal erneut ausführen, wenn Ihre Event-Funktion einen Kaltstart erlebt. Für solche Event-Funktionen können Sie sich dafür entscheiden, die Initialisierung einer bestimmten Funktion so lange zu verschieben, bis Ihre Event-Funktion sie benötigt. Sehen Sie sich zum Beispiel den folgenden Steuerungsablauf für einen Lambda-Handler an:

```
def handler(event, context):
    ...
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

Im vorherigen Beispiel entschied sich der Entwickler dagegen, `CLIENT_A` außerhalb des Haupthandlers zu initialisieren, und initialisierte ihn stattdessen innerhalb der `if`-Anweisung. Auf diese Weise führt Lambda diesen Code nur aus, wenn die Bedingung `some_condition` erfüllt wird. Wenn `CLIENT_A` außerhalb des Haupthandlers initialisiert wird, führt Lambda diesen Code bei jedem Kaltstart aus. Das kann die Gesamtlatenz erhöhen.

Sie können Kaltstarts messen, wenn Lambda skaliert, indem Sie Ihre Event-Funktion um eine X-Ray Überwachung erweitern. Eine Ereignisfunktion, die bereitgestellte Parallelität verwendet, zeigt kein Kaltstartverhalten, da die Ausführungsumgebung vor dem Aufruf vorbereitet wird. Die bereitgestellte Parallelität muss jedoch auf eine bestimmte Version oder einen bestimmten Alias einer Event-Funktion angewendet werden, nicht auf die $LATEST-Version. Falls weiterhin ein Kaltstartverhalten

auftritt, stellen Sie sicher, dass Sie die Alias-Version aufrufen, für die bereitgestellte Gleichzeitigkeit konfiguriert ist.

Verwendung von Umgebungsvariablen zur Anzeige und Steuerung des bereitgestellten Gleichzeitigkeitsverhaltens

Es ist möglich, dass Ihre Event-Funktion die gesamte bereitgestellte Parallelität verbraucht. Lambda verwendet On-Demand-Instances, um zusätzlichen Datenverkehr zu verarbeiten. Um festzustellen, welche Art von Initialisierung Lambda für eine bestimmte Umgebung verwendet hat, überprüfen Sie den Wert der Umgebungsvariable `AWS_LAMBDA_INITIALIZATION_TYPE`.

Diese Variable hat zwei mögliche Werte: `provisioned-concurrency` oder `on-demand`. Der Wert von `AWS_LAMBDA_INITIALIZATION_TYPE` ist unveränderlich und bleibt während der gesamten Lebensdauer der Umgebung konstant. Informationen zum Überprüfen des Werts einer Umgebungsvariablen in Ihrem Event-Funktionscode finden Sie unter. Abrufen von Lambda-Umgebungsvariablen

Wenn Sie die .NET 8-Laufzeit verwenden, können Sie die `AWS_LAMBDA_DOTNET_PREJIT` Umgebungsvariable konfigurieren, um die Latenz für Ereignisfunktionen zu verbessern, auch wenn sie keine bereitgestellte Parallelität verwenden. Die .NET-Laufzeitumgebung führt für jede Bibliothek, die der Code zum ersten Mal aufruft, eine verzögerte Kompilierung und Initialisierung durch. Daher kann der erste Aufruf einer Lambda-Ereignisfunktion länger dauern als nachfolgende. Um dies zu verhindern, können Sie einen von drei Werten für `AWS_LAMBDA_DOTNET_PREJIT` wählen:

- `ProvisionedConcurrency`: Lambda führt für alle Umgebungen, die bereitgestellte Gleichzeitigkeit verwenden, eine zeitliche JIT-Kompilierung durch. Dies ist der Standardwert.
- `Always`: Lambda führt für jede Umgebung eine JIT-Kompilierung im Voraus durch, auch wenn die Event-Funktion keine bereitgestellte Parallelität verwendet.
- `Never`: Lambda deaktiviert die JIT-Kompilierung für alle Umgebungen vorab.

Verstehen des Protokollierungs- und Abrechnungsverhaltens bei bereitgestellter Gleichzeitigkeit

Bei bereitgestellten Parallelitätsumgebungen wird der Initialisierungscode Ihrer Event Function während der Zuweisung und in regelmäßigen Abständen ausgeführt, wenn Lambda Instanzen Ihrer Umgebung recycelt. Lambda rechnet die Initialisierung auch dann ab, wenn die Umgebungs-Instance niemals eine Anforderung verarbeitet. Die bereitgestellte Gleichzeitigkeit wird laufend ausgeführt und

getrennt von den Initialisierungs- und Aufrufkosten in Rechnung gestellt. Weitere Einzelheiten finden Sie unter AWS Lambda -Preise.

Wenn Sie eine Lambda-Ereignisfunktion mit bereitgestellter Parallelität konfigurieren, initialisiert Lambda diese Ausführungsumgebung vor, sodass sie vor Aufrufanforderungen verfügbar ist. Lambda protokolliert bei jeder Initialisierung der Umgebung das Feld Init Duration der Event Function in einem Platform-InitReport-Protokollereignis im JSON-Protokollierungsformat. Um dieses Protokollereignis zu sehen, konfigurieren Sie Ihre JSON-Protokollebene auf mindestens `INFO`. Sie können die Telemetrie-API auch verwenden, um Plattformereignisse zu verarbeiten, bei denen das Feld „Initialisierungsdauer“ gemeldet wird.

Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung

Application Auto Scaling ermöglicht es Ihnen, die bereitgestellte Gleichzeitigkeit nach einem Zeitplan oder basierend auf der Auslastung zu verwalten. Wenn Ihre Event-Funktion vorhersehbare Verkehrsmuster empfängt, verwenden Sie die geplante Skalierung. Wenn Sie möchten, dass Ihre Event-Funktion einen bestimmten Nutzungsgrad beibehält, verwenden Sie eine Skalierungsrichtlinie für die Zielverfolgung.

> **Note**
>
> Wenn Sie Application Auto Scaling verwenden, um die bereitgestellte Parallelität Ihrer Event Function zu verwalten, stellen Sie sicher, dass Sie zuerst einen anfänglich bereitgestellten Parallelitätswert konfigurieren. Wenn Ihre Event Function keinen anfänglich bereitgestellten Parallelitätswert hat, verarbeitet Application Auto Scaling die Event Function-Skalierung möglicherweise nicht richtig.

Geplante Skalierung

Application Auto Scaling ermöglicht es Ihnen, Ihren eigenen Skalierungsplan entsprechend vorhersehbarer Laständerungen festzulegen. Weitere Informationen und Beispiele finden Sie unter Geplante Skalierung für Application Auto Scaling im Application Auto Scaling Scaling-Benutzerhandbuch und Scheduling AWS Lambda Provisioned Concurrency für wiederkehrende Spitzennutzung im AWS Compute-Blog.

Zielverfolgung

Mit der Zielverfolgung erstellt und verwaltet Application Auto Scaling eine Reihe von CloudWatch Alarmen, die darauf basieren, wie Sie Ihre Skalierungsrichtlinie definieren. Wenn diese Alarme aktiviert werden, passt Application Auto Scaling mithilfe der bereitgestellten Gleichzeitigkeit automatisch die Anzahl der zugewiesenen Umgebungen an. Verwenden Sie die Zielverfolgung für Anwendungen, die keine vorhersehbaren Auslastungsmuster aufweisen.

Verwenden Sie die API-Operationen `RegisterScalableTarget` und `PutScalingPolicy` Application Auto Scaling, um die bereitgestellte Gleichzeitigkeit mithilfe der Zielverfolgung zu skalieren. Wenn Sie beispielsweise die AWS Command Line Interface (CLI) verwenden, gehen Sie wie folgt vor:

1. Registrieren Sie den Alias einer Event-Funktion als Skalierungsziel. Im folgenden Beispiel wird der BLUE-Alias einer Event-Funktion mit dem Namen registriert`my-function`:

   ```
   aws application-autoscaling register-scalable-target --service-namespace lambda \
       --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
       --scalable-dimension lambda:function:ProvisionedConcurrency
   ```

2. Wenden Sie eine Skalierungsrichtlinie auf das Ziel an. Im folgenden Beispiel wird Application Auto Scaling so konfiguriert, dass die bereitgestellte Gleichzeitigkeitskonfiguration für einen Alias so angepasst wird, dass die Auslastung bei 70 % liegt. Sie können jedoch jeden beliebigen Wert zwischen 10 % und 90 % anwenden.

   ```
   aws application-autoscaling put-scaling-policy \
       --service-namespace lambda \
       --scalable-dimension lambda:function:ProvisionedConcurrency \
       --resource-id function:my-function:BLUE \
       --policy-name my-policy \
       --policy-type TargetTrackingScaling \
       --target-tracking-scaling-policy-configuration '{ "TargetValue":
    0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
    "LambdaProvisionedConcurrencyUtilization" }}'
   ```

Die Ausgabe sollte in etwa wie folgt aussehen:

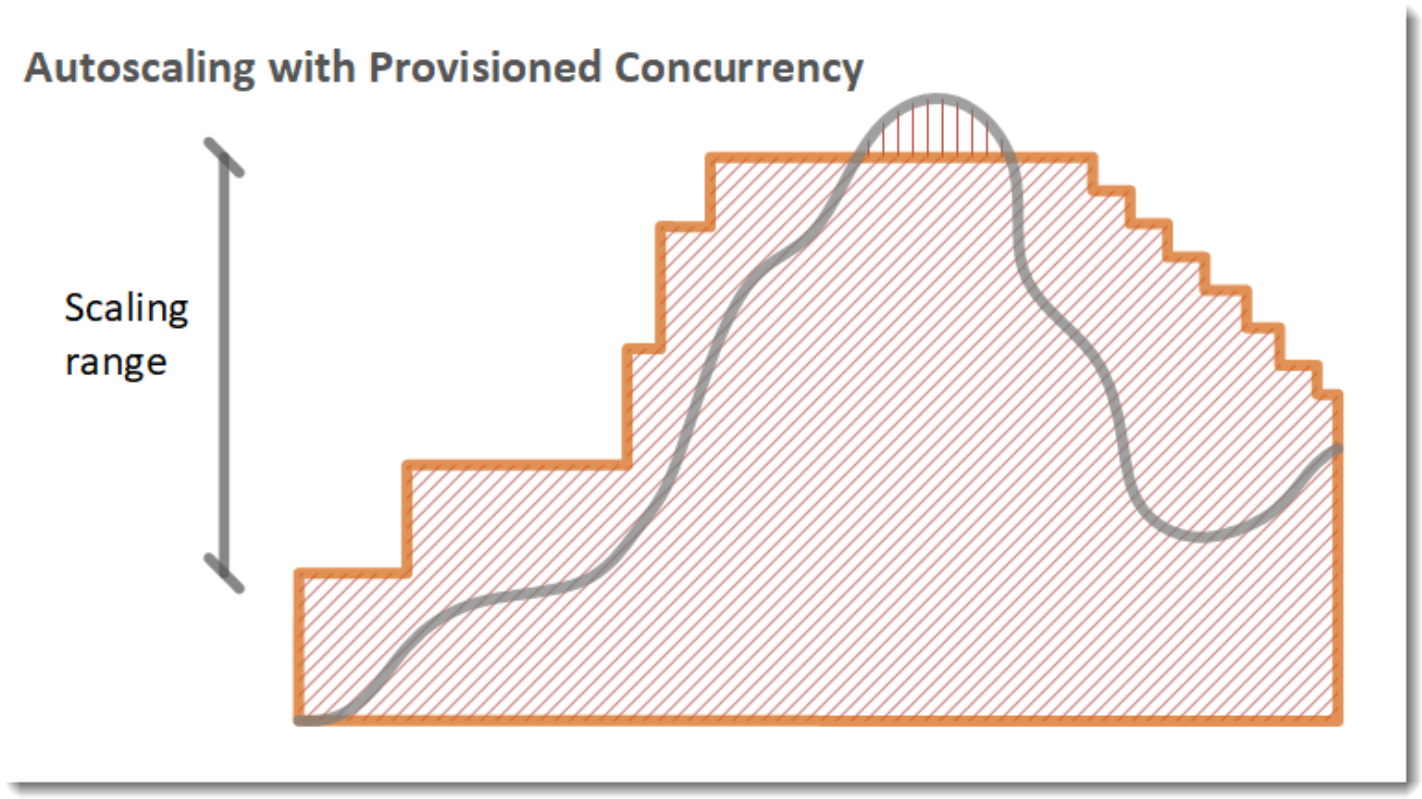
```
{
```

```
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

Application Auto Scaling erstellt zwei Alarme in CloudWatch. Der erste Alarm wird ausgelöst, wenn die Auslastung der bereitgestellten Gleichzeitigkeit konstant 70 % überschreitet. In diesem Fall weist Application Auto Scaling mehr bereitgestellte Gleichzeitigkeit zu, um die Auslastung zu reduzieren. Der zweite Alarm wird ausgelöst, wenn die Auslastung konstant unter 63 % (90 % des 70-%-Ziels) liegt. In diesem Fall reduziert Application Auto Scaling wird die bereitgestellte Gleichzeitigkeit des Alias.

> **Note**
>
> Lambda gibt die `ProvisionedConcurrencyUtilization` Metrik nur aus, wenn Ihre Event-Funktion aktiv ist und Anfragen empfängt. In Zeiten der Inaktivität werden keine Messwerte ausgegeben, und Ihre auto-scaling Skalierungsalarme wechseln in den `INSUFFICIENT_DATA` Status. Daher kann Application Auto Scaling die von Ihrer Event Function bereitgestellte Parallelität nicht anpassen. Dies kann zu einer unerwarteten Abrechnung führen.

Im folgenden Beispiel skaliert eine Event-Funktion je nach Auslastung zwischen einer Mindest- und Höchstmenge an bereitgestellter Parallelität.



Legende

- von Event-Funktionen
- Offene Anforderungen
- Bereitgestellte Concurrency
- Standard-Gleichzeitigkeit

Instanz

Wenn die Anzahl der offenen Anforderungen zunimmt, erhöht Application Auto Scaling die bereitgestellte Gleichzeitigkeit in großen Schritten, bis sie das konfigurierte Maximum erreicht. Danach kann die Event Function weiterhin mit der standardmäßigen, vorbehaltlosen Parallelität skaliert werden, falls Sie das Limit für die Parallelität Ihres Kontos noch nicht erreicht haben. Wenn die Auslastung sinkt und niedrig bleibt, verringert Application Auto Scaling die bereitgestellte Gleichzeitigkeit in kleineren periodischen Schritten.

Beide Alarme für Application Auto Scaling verwenden standardmäßig die Durchschnittsstatistik. Funktionen, bei denen es zu schnellen Verkehrsspitzen kommt, lösen diese Alarme möglicherweise nicht aus. Nehmen wir zum Beispiel an, dass Ihre Lambda-Ereignisfunktion schnell (d. h. 20-100 ms) ausgeführt wird und Ihr Datenverkehr in schnellen Ausbrüchen erfolgt. In diesem Fall übersteigt die Anzahl der Anfragen die zugewiesene, bereitgestellte Gleichzeitigkeit während eines Bursts.

Bei Application Auto Scaling muss die Burst-Load jedoch mindestens 3 Minuten lang aufrechterhalten werden, um zusätzliche Umgebungen bereitzustellen. Darüber hinaus benötigen beide CloudWatch Alarme 3 Datenpunkte, die den Zieldurchschnitt erreichen, um die Auto Scaling-Richtlinie zu aktivieren. Wenn Ihre Event-Funktion schnell stark ausgelastet ist, kann die Verwendung der Maximum-Statistik anstelle der Durchschnittsstatistik effektiver sein, um die bereitgestellte Parallelität zu skalieren und so Kaltstarts zu minimieren.

Weitere Informationen zu Zielverfolgungs-Skalierungsrichtlinien finden Sie unter Zielverfolgungs-Skalierungsrichtlinien für das Application Auto Scaling.

Skalierungsverhalten der Event-Funktion

Wenn Ihre Event-Funktion mehr Anfragen erhält, skaliert Lambda automatisch die Anzahl der Ausführungsumgebungen, um diese Anfragen zu bearbeiten, bis Ihr Konto seine Parallelitätsquote erreicht hat. Zum Schutz vor einer Überskalierung als Reaktion auf plötzliche Datenverkehrsspitzen begrenzt Lambda jedoch, wie schnell Ihre Funktionen skaliert werden können. Diese Skalierungsrate für Parallelität ist die maximale Geschwindigkeit, mit der Event Functions in Ihrem Konto als Reaktion auf erhöhte Anfragen skaliert werden können. (Das heißt, sie gibt an, wie schnell Lambda neue Ausführungsumgebungen erstellen kann.) Die Skalierungsrate für Parallelität unterscheidet sich vom Parallelitätslimit auf Kontoebene, das der Gesamtmenge an Parallelität entspricht, die Ihren Eventfunktionen zur Verfügung steht.

Nebenläufigkeitsskalierungsrate

In jeder AWS-Region und für jede Event-Funktion beträgt Ihre Parallelitätsskalierungsrate 1.000 Instanzen der Ausführungsumgebung alle 10 Sekunden (oder 10.000 Anfragen pro Sekunde alle 10 Sekunden). Mit anderen Worten: Lambda kann jeder Ihrer Funktionen alle 10 Sekunden höchstens 1 000 zusätzliche Instanzen der Ausführungsumgebung zuweisen oder 10 000 zusätzliche Anfragen pro Sekunde bearbeiten.

Normalerweise müssen Sie sich über diese Einschränkung keine Gedanken machen. Die Skalierungsrate von Lambda ist für die meisten Anwendungsfälle ausreichend.

Wichtig ist, dass die Skalierungsrate für die Gleichzeitigkeit eine Grenze auf Funktionsebene darstellt. Das bedeutet, dass jede Event-Funktion in Ihrem Konto unabhängig von anderen Event-Funktionen skaliert werden kann.

> **Note**
>
> In der Praxis versucht Lambda, Ihre Nebenläufigkeitsskalierungsrate kontinuierlich im Laufe der Zeit aufzufüllen, anstatt alle 10 Sekunden eine einzige Nachfüllung von 1 000 Einheiten durchzuführen.

Lambda sammelt keine ungenutzten Teile Ihrer Nebenläufigkeitsskalierungsrate an. Das bedeutet, dass Ihre Skalierungsrate zu jedem Zeitpunkt immer maximal 1 000 Nebenläufigkeitseinheiten beträgt. Wenn Sie beispielsweise in einem 10-Sekunden-Intervall keine der verfügbaren 1 000 Nebenläufigkeitseinheiten verwenden, werden Sie im nächsten 10-Sekunden-Intervall keine

1 000 zusätzlichen Einheiten ansammeln. Ihre Skalierungsrate für Nebenläufigkeit liegt im nächsten 10-Sekunden-Intervall immer noch bei 1 000.

Solange Ihre Event-Funktion weiterhin eine steigende Anzahl von Anfragen erhält, skaliert Lambda mit der schnellsten Rate, die Ihnen zur Verfügung steht, bis zum Parallelitätslimit Ihres Kontos. Sie können die Anzahl der Parallelität einschränken, die einzelne Event-Funktionen nutzen können, indem Sie die reservierte Parallelität konfigurieren. Wenn Anfragen schneller eingehen, als Ihre Event-Funktion skalieren kann, oder wenn Ihre Event-Funktion die maximale Parallelität erreicht, schlagen weitere Anfragen mit einem Drosselungsfehler fehl (Statuscode 429).

Überwachen der Gleichzeitigkeit

Lambda gibt CloudWatch Amazon-Metriken aus, um Sie bei der Überwachung der Parallelität Ihrer Lambda-Ereignisfunktionen zu unterstützen. In diesem Thema werden diese Metriken und ihre Interpretation erläutert.

Sections

- Allgemeine Metriken für Gleichzeitigkeit
- Metriken für Provisioned Concurrency
- Die Metrik ClaimedAccountConcurrency

Allgemeine Metriken für Gleichzeitigkeit

Verwenden Sie die folgenden Metriken, um die Parallelität Ihrer Lambda-Ereignisfunktionen zu überwachen. Die Granularität für jede Metrik ist 1 Minute.

- `ConcurrentExecutions` – Die Anzahl der aktiven gleichzeitigen Aufrufe zu einem bestimmten Zeitpunkt. Lambda gibt diese Metrik für alle Eventfunktionen, Versionen und Aliase aus. Für jede Event-Funktion in der Lambda-Konsole zeigt Lambda das Diagramm **`ConcurrentExecutions`** nativ auf der Registerkarte Überwachung unter Metriken an. Sehen Sie sich diese Metrik mit MAX an.
- `UnreservedConcurrentExecutions` – Die Anzahl der aktiven gleichzeitigen Aufrufe, die nicht reservierte Gleichzeitigkeit verwenden. Lambda gibt diese Metrik für alle Event-Funktionen in einer Region aus. Sehen Sie sich diese Metrik mit MAX an.
- `ClaimedAccountConcurrency` – Die Menge an Gleichzeitigkeit, die für On-Demand-Aufrufe nicht verfügbar ist. `ClaimedAccountConcurrency` entspricht `UnreservedConcurrentExecutions` plus dem Betrag der zugewiesenen Gleichzeitigkeit (d. h. der gesamten reservierten Gleichzeitigkeit plus der gesamten bereitgestellten Gleichzeitigkeit). Wenn das Kontogleichzeitigkeitslimit für `ClaimedAccountConcurrency` überschritten wird, können Sie ein höheres Kontogleichzeitigkeitslimit beantragen.

  Sehen Sie sich diese Metrik mit MAX an. Weitere Informationen finden Sie unter Die Metrik `ClaimedAccountConcurrency`.

Metriken für Provisioned Concurrency

Verwenden Sie die folgenden Metriken, um Lambda-Ereignisfunktionen mithilfe der bereitgestellten Parallelität zu überwachen. Die Granularität für jede Metrik ist 1 Minute.

- `ProvisionedConcurrentExecutions` – Die Anzahl der Instances der Ausführungsumgebung, die einen Aufruf bei bereitgestellter Gleichzeitigkeit aktiv verarbeiten. Lambda gibt diese Metrik für jede Version und jeden Alias der Event Function mit konfigurierter bereitgestellter Parallelität aus. Sehen Sie sich diese Metrik mit MAX an.

`ProvisionedConcurrentExecutions` entspricht nicht der Gesamtzahl der bereitgestellten Gleichzeitigkeit, die Sie zuweisen. Nehmen wir beispielsweise an, Sie weisen einer Version von Event Function 100 Einheiten bereitgestellter Parallelität zu. Wenn in einer bestimmten Minute höchstens 50 dieser 100 Ausführungsumgebungen gleichzeitig Aufrufe verarbeiten, dann ist der Wert von MAX(`ProvisionedConcurrentExecutions`) 50.

- `ProvisionedConcurrencyInvocations`— Die Häufigkeit, mit der Lambda Ihren Event-Funktionscode mithilfe der bereitgestellten Parallelität aufruft. Lambda gibt diese Metrik für jede Version und jeden Alias der Event Function mit konfigurierter bereitgestellter Parallelität aus. Sehen Sie sich diese Metrik mit SUM an.

`ProvisionedConcurrencyInvocations` unterscheidet sich von `ProvisionedConcurrentExecutions` dadurch, dass `ProvisionedConcurrencyInvocations` die Gesamtzahl der Aufrufe zählt, während `ProvisionedConcurrentExecutions` die Anzahl der aktiven Umgebungen zählt. Um diesen Unterschied zu verstehen, betrachten Sie das folgende Szenario:



In diesem Beispiel nehmen wir an, dass Sie 1 Aufruf pro Minute erhalten und jeder Aufruf 2 Minuten dauert. Jeder orangefarbene horizontale Balken steht für eine einzelne Anfrage. Nehmen wir an, Sie weisen dieser Ereignisfunktion 10 bereitgestellte Parallelitätseinheiten zu, sodass jede Anforderung mit bereitgestellter Parallelität ausgeführt wird.

Zwischen den Minuten 0 und 1 trifft `Request 1` ein. Bei Minute 1 ist der Wert für MAX(`ProvisionedConcurrentExecutions`) 1, da in der vergangenen Minute höchstens 1 Ausführungsumgebung aktiv war. Der Wert für SUM(`ProvisionedConcurrencyInvocations`) ist ebenfalls 1, da in der letzten Minute 1 neue Anforderung eingegangen ist.

Zwischen den Minuten 1 und 2 trifft `Request 2` ein, während `Request 1` weiterläuft. Bei Minute 2 ist der Wert für MAX(`ProvisionedConcurrentExecutions`) 2, da in der letzten Minute höchstens 2 Ausführungsumgebungen aktiv waren. Der Wert für SUM(`ProvisionedConcurrencyInvocations`) ist jedoch 1, da in der letzten Minute nur 1 neue Anforderung eingegangen ist. Dieses Verhalten der Metrik setzt sich bis zum Ende des Beispiels fort.

- `ProvisionedConcurrencySpilloverInvocations`— Die Häufigkeit, mit der Lambda Ihre Event-Funktion bei standardmäßiger (reservierter oder nicht reservierter) Parallelität aufruft, wenn die gesamte bereitgestellte Parallelität verwendet wird. Lambda gibt diese Metrik für jede Version und jeden Alias der Event Function mit konfigurierter bereitgestellter Parallelität aus. Sehen Sie sich diese Metrik mit SUM an. Der Wert von `ProvisionedConcurrencyInvocations` + `ProvisionedConcurrencySpilloverInvocations` sollte der Gesamtzahl der Aufrufe der Event-Funktion (d. h. der Metrik) entsprechen. `Invocations`

  `ProvisionedConcurrencyUtilization` – Der prozentuale Anteil der bereitgestellten Gleichzeitigkeit, der genutzt wird (d. h. der Wert von `ProvisionedConcurrentExecutions`

geteilt durch die Gesamtmenge der bereitgestellten Gleichzeitigkeit). Lambda gibt diese Metrik für jede Version und jeden Alias der Event Function mit konfigurierter bereitgestellter Parallelität aus. Sehen Sie sich diese Metrik mit MAX an.

Nehmen wir beispielsweise an, Sie stellen 100 Einheiten bereitgestellter Parallelität für eine Event Function-Version bereit. Wenn in einer bestimmten Minute höchstens 60 dieser 100 Ausführungsumgebungen gleichzeitig Aufrufe verarbeiten, dann ist der Wert von MAX(`ProvisionedConcurrentExecutions`) 60 und der Wert von MAX(`ProvisionedConcurrencyUtilization`) ist 0,6.

Ein hoher Wert für `ProvisionedConcurrencySpilloverInvocations` könnte bedeuten, dass Sie Ihrer Event Function zusätzliche bereitgestellte Parallelität zuweisen müssen. Alternativ können Sie Application Auto Scaling konfigurieren, um die automatische Skalierung der bereitgestellten Gleichzeitigkeit auf der Grundlage von vordefinierten Schwellenwerten zu steuern.

Umgekehrt `ProvisionedConcurrencyUtilization` könnten konstant niedrige Werte für darauf hindeuten, dass Sie Ihrer Event-Funktion zu viel bereitgestellte Parallelität zugewiesen haben.

Die Metrik **`ClaimedAccountConcurrency`**

Lambda verwendet die Metrik `ClaimedAccountConcurrency`, um zu ermitteln, wie viel Gleichzeitigkeit in Ihrem Konto für On-Demand-Aufrufe verfügbar ist. Lambda berechnet `ClaimedAccountConcurrency` mit der folgenden Formel:

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

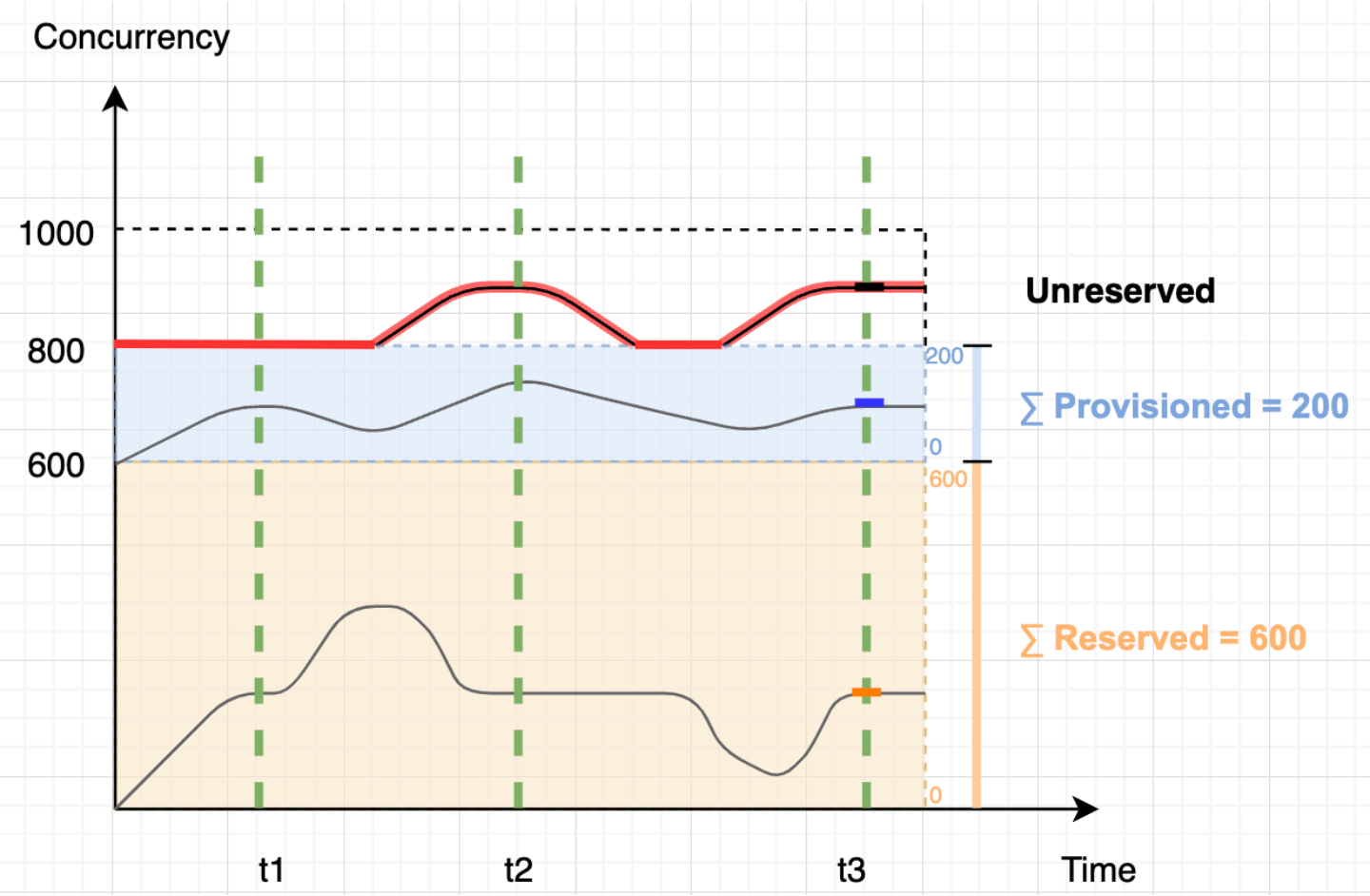
`UnreservedConcurrentExecutions` ist die Anzahl der aktiven gleichzeitigen Aufrufe, die nicht reservierte Gleichzeitigkeit verwenden. Die zugewiesene Gleichzeitigkeit ist die Summe der folgenden beiden Teile (wobei RC für „reservierte Gleichzeitigkeit“ und PC für „bereitgestellte Gleichzeitigkeit“ steht):

- Die Summe aller Event-Funktionen RC in einer Region.
- Die Summe PC aller Ereignisfunktionen in einer Region, die verwendenPC, mit Ausnahme von Ereignisfunktionen, die verwendenRC.

> **Note**
>
> Sie können nicht mehr PC als RC einer Ereignisfunktion zuweisen. Daher RC ist der Wert einer Ereignisfunktion immer größer oder gleich seinerPC. Um den Beitrag zur zugewiesenen Parallelität für solche Event-Funktionen PC sowohl mit als auch zu berechnenRC, berücksichtigt Lambda nurRC, welches das Maximum der beiden ist.

Lambda verwendet die Metrik `ClaimedAccountConcurrency` anstelle von `ConcurrentExecutions`, um zu ermitteln, wie viel Gleichzeitigkeit für On-Demand-Aufrufe verfügbar ist. Die Metrik `ConcurrentExecutions` ist zwar nützlich, um die Anzahl der aktiven gleichzeitigen Aufrufe nachzuverfolgen, spiegelt jedoch nicht immer die tatsächliche Verfügbarkeit von Gleichzeitigkeit wider. Das liegt daran, dass Lambda bei der Bestimmung der Verfügbarkeit auch die reservierte und die bereitgestellte Gleichzeitigkeit berücksichtigt.

Stellen Sie sich zur Veranschaulichung ein Szenario vor`ClaimedAccountConcurrency`, in dem Sie viele reservierte Parallelität und bereitgestellte Parallelität für Ihre Eventfunktionen konfigurieren, die weitgehend ungenutzt bleiben. Gehen Sie im folgenden Beispiel davon aus, dass Ihr Konto-Parallelitätslimit 1.000 beträgt und dass Ihr Konto über zwei Hauptereignisfunktionen verfügt: und. `function-orange` `function-blue` Sie weisen `function-orange` 600 reservierte Gleichzeitigkeitseinheiten zu. Sie weisen `function-blue` 200 bereitgestellte Gleichzeitigkeitseinheiten zu. Nehmen wir an, dass Sie im Laufe der Zeit zusätzliche Event-Funktionen bereitstellen und dabei das folgende Datenverkehrsmuster beobachten:



Im vorherigen Diagramm stehen die schwarzen Linien für die tatsächliche Nutzung der Gleichzeitigkeit im Zeitverlauf, und die rote Linie für den Wert `ClaimedAccountConcurrency` im Zeitverlauf. In diesem Szenario `ClaimedAccountConcurrency` sind es mindestens 800, obwohl die tatsächliche Parallelitätsauslastung Ihrer Event-Funktionen gering ist. Das liegt daran, dass Sie insgesamt 800 Gleichzeitigkeitseinheiten für `function-orange` und `function-blue` zugewiesen haben. Aus Sicht von Lambda haben Sie diese Parallelität für die Nutzung „beansprucht“, sodass Ihnen effektiv nur noch 200 Parallelitätseinheiten für andere Event-Funktionen zur Verfügung stehen.

Für dieses Szenario beträgt die zugewiesene Gleichzeitigkeit in der `ClaimedAccountConcurrency`-Formel 800. Daraus können wir den Wert von `ClaimedAccountConcurrency` an verschiedenen Stellen im Diagramm ableiten:

- Bei `t1`, `ClaimedAccountConcurrency` ist der Wert 800 (800 + 0 `UnreservedConcurrentExecutions`).
- Bei `t2`, `ClaimedAccountConcurrency` ist der Wert 900 (800 + 100 `UnreservedConcurrentExecutions`).

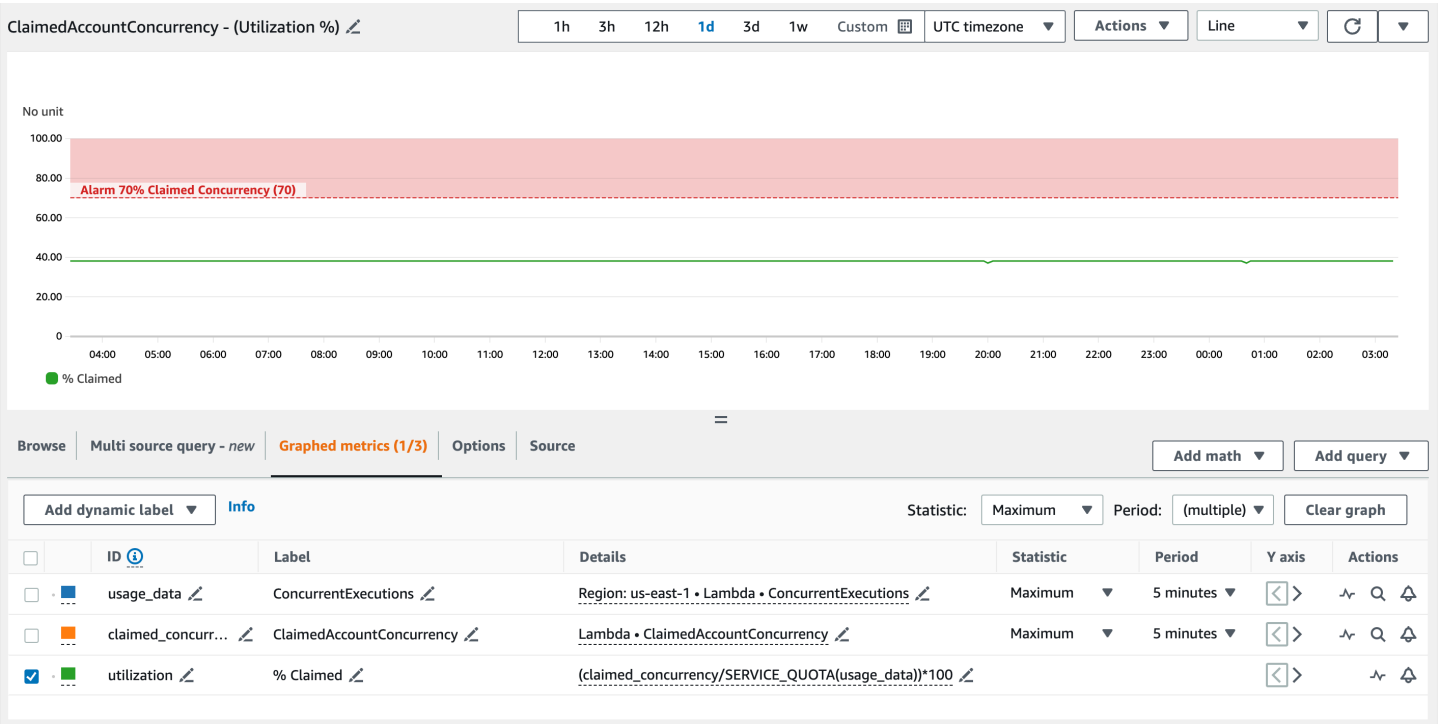
- Bei `t3`, `ClaimedAccountConcurrency` ist der Wert wieder 900 (800 + 100 `UnreservedConcurrentExecutions`).

Einrichtung der Metrik in **`ClaimedAccountConcurrency`** CloudWatch

Lambda gibt die `ClaimedAccountConcurrency` Metrik in aus. CloudWatch Verwenden Sie diese Metrik zusammen mit dem Wert von `SERVICE_QUOTA(ConcurrentExecutions)`, um die prozentuale Nutzung der Gleichzeitigkeit in Ihrem Konto zu ermitteln, wie in der folgenden Formel dargestellt:

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

Der folgende Screenshot zeigt, wie Sie diese Formel grafisch darstellen können. CloudWatch Die grüne `claim_utilization`-Linie steht für die Gleichzeitigkeitsauslastung in diesem Konto, die bei etwa 40 % liegt:



Der vorherige Screenshot enthält auch einen CloudWatch Alarm, der in den `ALARM` Status wechselt, wenn die Parallelitätsauslastung 70% übersteigt. Sie können die `ClaimedAccountConcurrency`-Metrik zusammen mit ähnlichen Alarmen verwenden, um proaktiv zu bestimmen, wann Sie möglicherweise ein höheres Kontengleichzeitigkeitslimit beantragen müssen.

Überwachung, Debuggen und Problembehandlung von Lambda-Ereignisfunktionen

AWS Lambda lässt sich mit anderen integrieren AWS-Services , um Sie bei der Überwachung und Fehlerbehebung Ihrer Lambda-Ereignisfunktionen zu unterstützen. Lambda überwacht Event Functions automatisch in Ihrem Namen und meldet Metriken über Amazon CloudWatch. Damit Sie Ihren Code während der Ausführung überwachen können, verfolgt Lambda automatisch die Anzahl der Anfragen, die Dauer des Aufrufs pro Anfrage und die Anzahl der Anfragen, die zu einem Fehler führen.

Sie können andere verwenden AWS-Services , um Probleme mit Ihren Lambda-Ereignisfunktionen zu beheben. In diesem Abschnitt wird beschrieben, wie Sie diese verwenden, AWS-Services um Ihre Lambda-Ereignisfunktionen und -anwendungen zu überwachen, zu verfolgen, zu debuggen und Fehler zu beheben. Einzelheiten zur Protokollierung von Event-Funktionen und zu Fehlern in jeder Laufzeit finden Sie in den einzelnen Laufzeitabschnitten. Zu den Themen gehören Anwendungsüberwachung, Anwendungssignale und die Protokollierung von Lambda-API-Aufrufen mit CloudTrail.

Sections

- Preisgestaltung
- CloudWatch Metriken mit Lambda verwenden
- Arbeiten mit Lambda Event Function-Logs
- Protokollierung AWS Lambda API-Aufrufe mit AWS CloudTrail
- Visualisieren Sie Aufrufe von Lambda-Ereignisfunktionen mit AWS X-Ray
- Überwachen Sie die Leistung von Eventfunktionen mit Amazon CloudWatch Lambda Insights
- Überwachen von Lambda-Anwendungen
- Überwachen Sie die Anwendungsleistung mit Amazon CloudWatch Application Signals
- Ferndebuggen von Lambda-Ereignisfunktionen mit Visual Studio Code

Preisgestaltung

CloudWatch hat ein unbefristetes kostenloses Kontingent. Bei Überschreitung des Schwellenwerts für das kostenlose Nutzungskontingent CloudWatch fallen Gebühren für Kennzahlen, Dashboards, Alarme, Protokolle und Erkenntnisse an. Weitere Informationen finden Sie unter CloudWatch Amazon-Preise.

CloudWatch Metriken mit Lambda verwenden

Wenn Ihre AWS Lambda Event-Funktion die Verarbeitung eines Ereignisses abgeschlossen hat, sendet Lambda automatisch Metriken über den Aufruf an Amazon. CloudWatch Sie müssen Ihrer Ausführungsrolle keine zusätzlichen Berechtigungen für den Empfang von Event Function-Metriken erteilen, und für diese Metriken fallen keine zusätzlichen Gebühren an.

Es gibt viele Arten von Metriken, die mit Lambda-Ereignisfunktionen verknüpft sind. Dazu gehören Aufrufmetriken, Leistungsmetriken, Gleichzeitigkeitsmetriken, Metriken für asynchrone Aufrufe und Metriken für die Zuordnung von Ereignisquellen. Weitere Informationen finden Sie unter the section called “Metriktypen”.

In der CloudWatch Konsole können Sie sich diese Metriken ansehen und daraus Diagramme und Dashboards erstellen. Sie können auch Alarme so einstellen, dass sie auf Änderungen bei Auslastung, Leistung oder Fehlerraten reagieren. Lambda sendet metrische Daten CloudWatch in 1-Minuten-Intervallen an.

Für einen direkteren Einblick in Ihre Lambda-Ereignisfunktion können Sie hochauflösende benutzerdefinierte Metriken erstellen. Für benutzerdefinierte Metriken und CloudWatch Alarme fallen Gebühren an. Weitere Informationen finden Sie unter CloudWatch Amazon-Preise.

Metriken für Lambda-Ereignisfunktionen anzeigen

Verwenden Sie die CloudWatch Konsole, um Metriken für Ihre Lambda Event Functions anzuzeigen. In der Konsole können Sie Funktionsmetriken nach Funktionsname, Alias, Version oder UUID der Zuordnung von Ereignisquellen filtern und sortieren.

Um Metriken auf der CloudWatch Konsole anzuzeigen

1. Öffnen Sie die Seite „Metriken“ (`AWS/Lambda`Namespace) der CloudWatch Konsole.
2. Wählen Sie auf der Registerkarte Durchsuchen unter Metriken eine der folgenden Dimensionen aus:

   - Nach Funktionsname (`FunctionName`) — Zeigt aggregierte Metriken für alle Versionen und Aliase einer Event-Funktion an.
   - Nach Ressource (`Resource`) — Metriken für eine Version oder einen Alias einer Event-Funktion anzeigen.
   - Nach ausgeführter Version (`ExecutedVersion`) – Zeigen Sie Metriken für eine Kombination aus Alias und Version an. Verwenden Sie die `ExecutedVersion` Dimension, um die

Fehlerraten für zwei Versionen einer Ereignisfunktion zu vergleichen, die beide Ziele eines gewichteten Alias sind.

- Nach Zuordnung von Ereignisquellen UUID (`EventSourceMappingUUID`) — Metriken für eine Zuordnung von Ereignisquellen anzeigen.
- Funktionsübergreifend (keine) — Zeigt aggregierte Metriken für alle aktuellen Ereignisfunktionen an AWS-Region.

3. Wählen Sie eine Metrik. Die Metrik sollte automatisch im visuellen Diagramm sowie auf der Registerkarte Graphische Metriken angezeigt werden.

Standardmäßig verwenden Diagramme die `Sum`-Statistik für alle Metriken. Um eine andere Statistik auszuwählen und das Diagramm anzupassen, verwenden Sie die Optionen auf der Registerkarte Graphed metrics (Graphierte Metriken).

> **Note**
>
> Der Zeitstempel einer Metrik gibt an, wann die Ereignisfunktion aufgerufen wurde. Abhängig von der Dauer des Aufrufs kann es einige Minuten dauern, bis die Metrik ausgegeben wird. Wenn Ihre Event-Funktion beispielsweise ein Timeout von 10 Minuten hat, suchen Sie nach genauen Messwerten, die mehr als 10 Minuten in der Vergangenheit zurückliegen.

Weitere Informationen zu CloudWatch finden Sie im CloudWatch Amazon-Benutzerhandbuch.

Arten von Metriken für Lambda-Ereignisfunktionen

In diesem Abschnitt werden die Typen von Lambda-Metriken beschrieben, die in der CloudWatch Konsole verfügbar sind.

Themen

- Aufrufmetriken
- Bereitstellungsmetriken
- Leistungsmetriken
- Gleichzeitigkeitsmetriken
- Metriken asynchroner Aufrufe
- Metriken zur Zuordnung von Ereignisquellen

Aufrufmetriken

Aufrufmetriken sind binäre Indikatoren für das Ergebnis eines Aufrufs einer Lambda-Ereignisfunktion. Betrachten Sie diese Metriken mit der Statistik `Sum`. Wenn die Event-Funktion beispielsweise einen Fehler zurückgibt, sendet Lambda die `Errors` Metrik mit dem Wert 1. Um die Anzahl der Fehler in der Ereignisfunktion zu ermitteln, die jede Minute aufgetreten sind, sehen Sie sich die `Sum` `Errors` Metrik mit einem Zeitraum von 1 Minute an.

- `Invocations`— Die Häufigkeit, mit der Ihr Event-Funktionscode aufgerufen wurde, einschließlich erfolgreicher Aufrufe und Aufrufe, die zu einem Funktionsfehler führen. Aufrufe werden nicht aufgezeichnet, wenn die Aufrufanforderung gedrosselt wird oder anderweitig zu einem Aufruffehler führt. Der Wert von `Invocations` entspricht der Anzahl der fakturierten Anforderungen.
- `Errors` – Die Anzahl der Aufrufe, die zu einem Funktionsfehler führen. Funktionsfehler schließen Ausnahmen ein, die Ihr Code ausgibt, und Ausnahmen, die die Lambda-Laufzeit ausgibt. Die Laufzeit gibt Fehler für Probleme wie Timeouts und Konfigurationsfehler zurück. Um die Fehlerquote zu berechnen, teilen Sie den Wert von `Errors` durch den Wert von `Invocations`. Beachten Sie, dass der Zeitstempel einer Fehlermetrik angibt, wann die Event-Funktion aufgerufen wurde, und nicht, wann der Fehler aufgetreten ist.
- `DeadLetterErrors`— Bei asynchronen Aufrufen die Häufigkeit, mit der Lambda versucht, ein Ereignis an eine Dead-Letter-Warteschlange (DLQ) zu senden, aber fehlschlägt. Dead-letter Fehler können aufgrund falsch eingestellter Ressourcen oder Größenbeschränkungen auftreten.
- `DestinationDeliveryFailures` – Bei asynchronen Aufrufen und unterstützten Zuordnung von Ereignisquellen die Häufigkeit, mit der Lambda versucht, ein Ereignis an ein Ziel zu senden, dies jedoch fehlschlägt. Für die Zuordnung von Ereignisquellen unterstützt Lambda Ziele für Streamquellen (DynamoDB und Kinesis). Zustellungsfehler können aufgrund von Berechtigungsfehlern, falsch konfigurierten Ressourcen oder Größenbeschränkungen auftreten. Fehler können auch dann passieren, wenn Ihre Konfiguration einen nicht unterstützten Zieltyp enthält, z. B. eine Amazon-SQS-FIFO-Warteschlange oder ein Amazon-SNS-FIFO-Thema.
- `Throttles` – Die Anzahl der Aufruf-Anforderungen, die gedrosselt werden. Wenn alle Funktions-Instances Anforderungen verarbeiten und keine Nebenläufigkeit zum Hochskalieren verfügbar ist, lehnt Lambda zusätzliche Anforderungen mit einem `TooManyRequestsException`-Fehler ab. Gedrosselte Anforderungen und andere Aufruffehler zählen nicht als `Invocations` oder `Errors`.

> **Note**
>
> Mit Lambda Managed Instances bietet Lambda detaillierte Drosselungsmetriken, die die spezifische Einschränkung identifizieren, die die Drosselung verursacht hat. Wenn

in der Ausführungsumgebung eine Drosselung auftritt, wird genau eine der folgenden Untermetriken mit dem Wert 1 ausgegeben, während die restlichen drei mit dem Wert 0 ausgegeben werden. Die `Throttles` Metrik wird immer zusammen mit diesen Untermetriken ausgegeben.

- `CPUThrottles`— Aufrufe wurden aufgrund von CPU-Erschöpfung in der Ausführungsumgebung gedrosselt.
- `MemoryThrottles`— Aufrufe wurden aufgrund von Speichererschöpfung in der Ausführungsumgebung gedrosselt.
- `DiskThrottles`— Aufrufe wurden aufgrund von Festplattenauslastung in der Ausführungsumgebung gedrosselt.
- `ConcurrencyThrottles`— Aufrufe werden gedrosselt, wenn das Parallelitätslimit der Ausführungsumgebung erreicht ist.

- `OversizedRecordCount`— Für Amazon DocumentDB DocumentDB-Ereignisquellen die Anzahl der Ereignisse, die Ihre Event-Funktion von Ihrem Change-Stream empfängt und die eine Größe von mehr als 6 MB haben. Lambda verwirft die Nachricht und gibt diese Metrik aus.
- `ProvisionedConcurrencyInvocations`— Die Häufigkeit, mit der Ihr Event-Funktionscode mithilfe der bereitgestellten Parallelität aufgerufen wird.
- `ProvisionedConcurrencySpilloverInvocations`— Die Häufigkeit, mit der Ihr Event-Funktionscode mithilfe der Standardparallelität aufgerufen wird, wenn die gesamte bereitgestellte Parallelität verwendet wird.
- `RecursiveInvocationsDropped`— Die Häufigkeit, mit der Lambda den Aufruf Ihrer Event-Funktion gestoppt hat, weil es erkannt hat, dass Ihre Event-Funktion Teil einer unendlichen rekursiven Schleife ist. Die rekursive Schleifenerkennung überwacht, wie oft eine Event-Funktion als Teil einer Kette von Anfragen aufgerufen wird, indem sie Metadaten verfolgt, die von unterstützten SDKs hinzugefügt wurden. AWS Wenn Ihre Event-Funktion als Teil einer Kette von Anfragen etwa 16 Mal aufgerufen wird, verwirft Lambda standardmäßig den nächsten Aufruf. Wenn Sie die rekursive Schleifenerkennung deaktivieren, wird diese Metrik nicht ausgegeben. Weitere Informationen über dieses Feature finden Sie unter Verwenden Sie die rekursive Lambda-Schleifenerkennung, um Endlosschleifen zu verhindern.

Bereitstellungsmetriken

Bereitstellungsmetriken liefern Informationen über Lambda Event Function-Bereitstellungsereignisse und zugehörige Validierungsprozesse.

- `SignatureValidationErrors` – Gibt an, wie oft bei der Bereitstellung eines Codepakets Fehler bei der Signaturvalidierung aufgetreten sind, obwohl die Codesignatur-Konfigurationsrichtlinie auf `Warn` festgelegt ist. Diese Metrik wird ausgegeben, wenn die Prüfung auf Ablauf, Diskrepanz oder Widerruf scheitert, die Bereitstellung jedoch aufgrund der `Warn`-Richtlinieneinstellung weiterhin zulässig ist. Weitere Informationen zur Codesignatur finden Sie unter Verwendung von Codesignatur zur Überprüfung der Codeintegrität mit Lambda.

Leistungsmetriken

Leistungskennzahlen liefern Leistungsdetails zu einem einzelnen Aufruf der Event Function. Die `Duration` Metrik gibt beispielsweise die Zeit in Millisekunden an, die Ihre Event-Funktion mit der Verarbeitung eines Ereignisses verbringt. Um einen Eindruck davon zu bekommen, wie schnell Ihre Event-Funktion Ereignisse verarbeitet, sehen Sie sich diese Metriken zusammen mit der `Average` Statistik oder an. `Max`

- `Duration`— Die Zeit, die Ihr Event-Funktionscode mit der Verarbeitung eines Ereignisses verbringt. Die fakturierte Dauer für einen Aufruf ist der Wert von `Duration`, aufgerundet auf die nächste Millisekunde. `Duration` beinhaltet nicht die Kaltstartzeit.
- `PostRuntimeExtensionsDuration`— Die Gesamtdauer, die die Runtime mit der Ausführung von Code für Erweiterungen verbringt, nachdem der Event-Funktionscode abgeschlossen ist.
- `IteratorAge` – Bei DynamoDB-, Kinesis- und Amazon-DocumentDB-Ereignisquellen das Alter des letzten Datensatzes im Ereignis in Millisekunden. Diese Metrik misst die Zeit zwischen dem Empfang des Datensatzes durch einen Stream und dem Senden des Ereignisses durch die Zuordnung der Ereignisquelle an die Ereignisfunktion.
- `OffsetLag`— Bei selbstverwalteten Apache Kafka- und Amazon Managed Streaming for Apache Kafka (Amazon MSK) -Ereignisquellen der Unterschied im Offset zwischen dem letzten Datensatz, der in ein Thema geschrieben wurde, und dem letzten Datensatz, den die Nutzergruppe Ihrer Event-Funktion verarbeitet hat. Ein Kafka-Thema kann zwar mehrere Partitionen haben, diese Metrik misst die Offset-Verzögerung jedoch auf Themenebene.

`Duration` unterstützt auch Perzentilstatistiken (p). Verwenden Sie Perzentile, um Ausreißerwerte auszuschließen, die `Averages`- und `Maximum`-Statistiken verzerren. Die p95-Statistik zeigt beispielsweise die maximale Dauer von 95 % der Aufrufe, ohne die langsamsten 5 %. Weitere Informationen finden Sie unter Perzentile im CloudWatch Amazon-Benutzerhandbuch.

Gleichzeitigkeitsmetriken

Lambda meldet Parallelitätsmetriken als aggregierte Anzahl der Instanzen, die Ereignisse in einer Event-Funktion, Version, einem Alias oder verarbeiten. AWS-Region Zeigen Sie diese Metriken mit der Statistik `Max` an, um zu sehen, wie nah Sie an den Grenzwerten für Gleichzeitigkeit sind.

- `ConcurrentExecutions`— Die Anzahl der Event Function-Instanzen, die Ereignisse verarbeiten. Wenn diese Zahl Ihr Kontingent für gleichzeitige Ausführungen für die Region oder das reservierte Parallelitätslimit für die Event Function erreicht, drosselt Lambda zusätzliche Aufrufanforderungen.
- `ProvisionedConcurrentExecutions`— Die Anzahl der Event Function-Instanzen, die Ereignisse mithilfe der bereitgestellten Parallelität verarbeiten. Für jeden Aufruf eines Alias oder einer Version mit Provisioned Concurrency gibt Lambda die aktuelle Anzahl aus. Wenn Ihre Event-Funktion inaktiv ist oder keine Anfragen empfängt, gibt Lambda diese Metrik nicht aus.
- `ProvisionedConcurrencyUtilization` – Für eine Version oder einen Alias der Wert von `ProvisionedConcurrentExecutions` geteilt durch die Gesamtmenge der konfigurierten Gleichzeitigkeit. Wenn Sie beispielsweise eine bereitgestellte Parallelität von 10 für Ihre Event-Funktion konfigurieren und Ihr Wert 7 `ProvisionedConcurrentExecutions` ist, dann ist Ihr Wert 0,7. `ProvisionedConcurrencyUtilization`

  Wenn Ihre Event-Funktion inaktiv ist oder keine Anfragen empfängt, gibt Lambda diese Metrik nicht aus, da sie auf `ProvisionedConcurrentExecutions` basiert. Beachten Sie dies, wenn Sie es `ProvisionedConcurrencyUtilization` als Grundlage für CloudWatch Alarme verwenden.
- `UnreservedConcurrentExecutions`— Für eine Region die Anzahl der Ereignisse, die Event Functions ohne reservierte Parallelität verarbeiten.
- `ClaimedAccountConcurrency` – Mit Bezug auf eine Region die Menge an Gleichzeitigkeit, die für On-Demand-Aufrufe nicht verfügbar ist. `ClaimedAccountConcurrency` entspricht `UnreservedConcurrentExecutions` plus dem Betrag der zugewiesenen Gleichzeitigkeit (d. h. der gesamten reservierten Gleichzeitigkeit plus der gesamten bereitgestellten Gleichzeitigkeit). Weitere Informationen finden Sie unter Die Metrik `ClaimedAccountConcurrency`.

Metriken asynchroner Aufrufe

Metriken asynchroner Aufrufe liefern Details zu asynchronen Aufrufen aus Ereignisquellen und direkten Aufrufen. Sie können Schwellenwerte und Alarme festlegen, um Sie über bestimmte Änderungen zu informieren. Zum Beispiel, wenn die Anzahl der Ereignisse, die für die Verarbeitung in die Warteschlange gestellt werden, ungewollt ansteigt (`AsyncEventsReceived`). Oder wenn ein Ereignis lange auf seine Verarbeitung gewartet hat (`AsyncEventAge`).

- `AsyncEventsReceived` – Die Anzahl der Ereignisse, die Lambda erfolgreich zur Verarbeitung in die Warteschlange stellt. Diese Metrik gibt Aufschluss über die Anzahl der Ereignisse, die eine Event-Funktion empfängt. Überwachen Sie diese Metrik und stellen Sie Alarme für Schwellenwerte ein, um nach Problemen zu suchen. Zum Beispiel, um eine unerwünschte Anzahl von Ereignissen zu erkennen, die an Lambda gesendet werden, und um schnell Probleme zu diagnostizieren, die auf falsche Auslöser- oder Funktionskonfigurationen zurückzuführen sind. Diskrepanzen zwischen `AsyncEventsReceived` und `Invocations` können auf eine unterschiedliche Verarbeitung, verworfene Ereignisse oder einen möglichen Rückstand in der Warteschlange hinweisen.
- `AsyncEventAge`— Die Zeit zwischen dem erfolgreichen Einreihen des Ereignisses durch Lambda in die Warteschlange und dem Aufrufen der Event-Funktion. Der Wert dieser Metrik steigt, wenn Ereignisse aufgrund von Aufruffehlern oder Drosselung erneut versucht werden. Überwachen Sie diese Metrik und legen Sie Alarme für Schwellenwerte für verschiedene Statistiken fest, wenn sich eine Warteschlange immer länger wird. Um einen Anstieg dieser Metrik zu beheben, sehen Sie sich die `Errors`-Metrik an, um Funktionsfehler zu identifizieren, und die `Throttles`-Metrik, um Gleichzeitigkeitsprobleme zu identifizieren.
- `AsyncEventsDropped`— Die Anzahl der Ereignisse, die gelöscht werden, ohne dass die Event-Funktion erfolgreich ausgeführt wurde. Wenn Sie eine Warteschlange für unzustellbare Nachrichten oder ein `OnFailure`-Ziel konfigurieren, werden Ereignisse dorthin gesendet, bevor sie gelöscht werden. Ereignisse werden aus verschiedenen Gründen verworfen. Beispielsweise können Ereignisse das maximale Ereignisalter überschreiten oder die maximale Anzahl von Wiederholungsversuchen ausschöpfen, oder die reservierte Gleichzeitigkeit ist möglicherweise auf 0 gesetzt. Um herauszufinden, warum Ereignisse verworfen werden, sehen Sie sich die `Errors`-Metrik an, um Funktionsfehler zu identifizieren, und die `Throttles`-Metrik, um Gleichzeitigkeitsprobleme zu identifizieren.

Metriken zur Zuordnung von Ereignisquellen

Metriken zur Zuordnung von Ereignisquellen bieten Einblicke in das Verarbeitungsverhalten Ihrer Zuordnung von Ereignisquellen.

Derzeit sind Metriken zur Zuordnung von Ereignisquellen für Amazon SQS-, Kinesis-, DynamoDB-, Amazon MSK- und selbstverwaltete Apache Kafka-Ereignisquellen verfügbar.

Für die Zuordnung von Ereignisquellen mit Metrikkonfiguration können Sie jetzt auch alle ESM-bezogenen Metriken auf der Registerkarte Monitor auf der Seite Konsole Lambda **>** Zusätzliche Ressourcen **>** Ereignisquellenzuordnungen überprüfen.

So aktivieren Sie Metriken oder eine Zuordnung von Ereignisquellen (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, für die Sie Metriken aktivieren möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie die Zuordnung von Ereignisquellen, für die Sie Metriken aktivieren möchten und wählen Sie dann Bearbeiten.
5. Wählen Sie unter Konfiguration für die Zuordnung von Ereignisquellen die Option Metriken aktivieren aus oder wählen Sie eine Option aus der Dropdownliste Metriken aus.
6. Wählen Sie Speichern.

Alternativ können Sie Metriken für Ihre Ereignisquellenzuordnung programmgesteuert aktivieren, indem Sie das EventSourceMappingMetricsConfigObjekt in Ihrem. EventSourceMappingConfiguration Der folgende UpdateEventSourceMappingCLI-Befehl aktiviert beispielsweise Metriken für eine Ereignisquellenzuordnung:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --metrics-config Metrics=EventCount
```

Es gibt 3 Metrikgruppen:`EventCount`, `ErrorCount` und`KafkaMetrics`, und jede Gruppe hat mehrere Metriken. Nicht jede Metrik ist für jede Ereignisquelle verfügbar. In der folgenden Tabelle sind die unterstützten Metriken für jeden Typ von Ereignisquelle zusammengefasst.

Sie müssen sich für die Metrikgruppe anmelden, um metrikbezogene Metriken zu erhalten. Legen Sie beispielsweise EventCount in der Metrikkonfiguration Folgendes fest: (`PolledEventCount`,,`FilteredOutEventCount`,, `InvokedEventCount` `FailedInvokeEventCount``DroppedEventCount`, `OnFailureDestinationDeliveredEventCount` und). `DeletedEventCount`

Metrik zur Zuordnung von Ereignisquellen	Metrikgruppe	Amazon SQS	Kinesis- und DynamoDB-Streams	Amazon MSK und selbstver walteter Apache Kafka
PolledEventCount	EventCount	Ja	Ja	Ja
FilteredOutEventCount	EventCount	Ja	Ja	Ja
InvokedEventCount	EventCount	Ja	Ja	Ja
FailedInvokeEventCount	EventCount	Ja	Ja	Ja
DroppedEventCount	EventCount	Nein	Ja	Ja
OnFailureDestinationDeliveredEventCount	EventCount	Nein	Ja	Ja
DeletedEventCount	EventCount	Ja	Nein	Nein
CommittedEventCount	EventCount	Nein	Nein	Ja
PollingErrorCount	ErrorCount	Nein	Nein	Ja

Metrik zur Zuordnung von Ereignisquellen	Metrikgruppe	Amazon SQS	Kinesis- und DynamoDB-Streams	Amazon MSK und selbstverwalteter Apache Kafka
`InvokeErrorCount`	`ErrorCount`	Nein	Nein	Ja
`OnFailureDestinationDeliveryErrorCount`	`ErrorCount`	Nein	Nein	Ja
`SchemaRegistryErrorCount`	`ErrorCount`	Nein	Nein	Ja
`CommitErrorCount`	`ErrorCount`	Nein	Nein	Ja
`MaxOffsetLag`	`KafkaMetrics`	Nein	Nein	Ja
`SumOffsetLag`	`KafkaMetrics`	Nein	Nein	Ja

Wenn sich Ihre Zuordnung von Ereignisquellen im Bereitstellungsmodus befindet, stellt Lambda außerdem die folgende Metrik bereit:

- `ProvisionedPollers` – Bei Zuordnungen von Ereignisquellen im Bereitstellungsmodus die Anzahl der aktiv laufenden Ereignis-Poller. Sehen Sie sich diese Metrik mithilfe der Mathematik an. `MAX`
- (Nur Amazon MSK und selbstverwaltete Apache Kafka-Ereignisquellen) `EventPollerUnit` — Für Ereignisquellenzuordnungen im Bereitstellungsmodus die Anzahl der Event-Poller-Einheiten, die aktiv ausgeführt werden. Sehen Sie sich diese Metrik mithilfe der Mathematik an. `SUM`

- (Amazon MSK und selbstverwaltete Apache Kafka-Ereignisquellen) `EventPollerThroughputInBytes` — Für Ereignisquellenzuordnungen im Bereitstellungsmodus die gesamte Datensatzgröße der von der Ereignisquelle abgefragten Event-Poller. Es kann Ihnen den aktuellen Abfragedurchsatz mitteilen. Sehen Sie sich diese Metrik mithilfe der `SUM` Mathematik an.

Hier finden Sie weitere Informationen zu den einzelnen Metriken:

- `PolledEventCount` – Die Anzahl der Ereignisse, die Lambda erfolgreich aus der Ereignisquelle liest. Wenn Lambda nach Ereignissen abfragt, aber eine leere Umfrage erhält (keine neuen Datensätze), gibt Lambda einen Wert von 0 für diese Metrik aus. Verwenden Sie diese Metrik, um zu ermitteln, ob Ihre Zuordnung von Ereignisquellen korrekt nach neuen Ereignissen fragt.
- `FilteredOutEventCount` – Bei der Zuordnung von Ereignisquellen mit einem Filterkriterium die Anzahl der Ereignisse, die nach diesen Filterkriterien herausgefiltert wurden. Verwenden Sie diese Metrik, um festzustellen, ob Ihre Zuordnung von Ereignisquellen Ereignisse ordnungsgemäß herausfiltert. Für Ereignisse, die den Filterkriterien entsprechen, gibt Lambda eine 0-Metrik aus.
- `InvokedEventCount`— Die Anzahl der Ereignisse, die Ihre Lambda-Ereignisfunktion aufgerufen haben. Verwenden Sie diese Metrik, um zu überprüfen, ob Ereignisse Ihre Event-Funktion ordnungsgemäß aufrufen. Wenn ein Ereignis zu einem Funktionsfehler oder einer Drosselung führt, `InvokedEventCount` kann es aufgrund automatischer Wiederholungen zu mehreren Zählungen für dasselbe abgefragte Ereignis kommen.

  > **Warning**
  >
  > Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Aus diesem Grund können Ereignisse in Metriken, die Ereigniszählungen beinhalten, mehrfach gezählt werden.

- `FailedInvokeEventCount`— Die Anzahl der Ereignisse, mit denen Lambda versucht hat, Ihre Event-Funktion aufzurufen, aber fehlgeschlagen ist. Aufrufe können aus Gründen wie Netzwerkkonfigurationsproblemen, falschen Berechtigungen oder einer gelöschten Lambda-Ereignisfunktion, -version oder einem gelöschten Alias fehlschlagen. Wenn für Ihre Zuordnung von Ereignisquellen partielle Batch-Antworten aktiviert sind, enthält `FailedInvokeEventCount` jedes Ereignis mit einem nicht leeren `BatchItemFailures` in der Antwort.

> **Note**
>
> Der Zeitstempel für die `FailedInvokeEventCount` Metrik steht für das Ende des Aufrufs der Event-Funktion. Dieses Verhalten unterscheidet sich von anderen Lambda-Aufruffehlermetriken, die zu Beginn des Aufrufs der Ereignisfunktion mit einem Zeitstempel versehen werden.

- `DroppedEventCount` – Die Anzahl der Ereignisse, die Lambda aufgrund von Ablauf oder wiederholter Erschöpfung ausgab. Genauer gesagt ist dies die Anzahl der Datensätze, die die von Ihnen konfigurierten Werte für `MaximumRecordAgeInSeconds` oder `MaximumRetryAttempts` überschreiten. Wichtig ist, dass dies nicht die Anzahl der Datensätze beinhaltet, die aufgrund einer Überschreitung der Aufbewahrungseinstellungen Ihrer Ereignisquelle ablaufen. Verworfene Ereignisse schließen auch Ereignisse aus, die Sie bei einem Ausfall an ein Ziel senden. Verwenden Sie diese Metrik, um einen zunehmenden Rückstand an Ereignissen zu erkennen.
- `OnFailureDestinationDeliveredEventCount` – Bei Zuordnungen von Ereignisquellen mit konfiguriertem Ausfallziel, die Anzahl der Ereignisse, die an dieses Ziel gesendet wurden. Verwenden Sie diese Metrik, um Funktionsfehler im Zusammenhang mit Aufrufen von dieser Ereignisquelle zu überwachen. Wenn die Lieferung an das Ziel fehlschlägt, verarbeitet Lambda die Metriken wie folgt:
  - Lambda gibt die `OnFailureDestinationDeliveredEventCount`-Metrik nicht aus.
  - Für die `DestinationDeliveryFailures`-Metrik gibt Lambda eine 1 aus.
  - Für die `DroppedEventCount`-Metrik gibt Lambda eine Zahl aus, die der Anzahl der Ereignisse entspricht, bei denen die Übermittlung fehlgeschlagen ist.
- `DeletedEventCount` – Die Anzahl der Ereignisse, die Lambda nach der Verarbeitung erfolgreich löscht. Wenn Lambda versucht, ein Ereignis zu löschen, dies jedoch fehlschlägt, gibt Lambda eine 0-Metrik aus. Verwenden Sie diese Metrik, um sicherzustellen, dass erfolgreich verarbeitete Ereignisse aus Ihrer Ereignisquelle gelöscht werden.
- `CommittedEventCount`— Die Anzahl der Ereignisse, die Lambda nach der Verarbeitung erfolgreich festgeschrieben hat. Es handelt sich um eine Summe der Deltas des letzten und des aktuellen Offsets, das von jeder Partition in der Kafka-Ereignisquellenzuordnung vorgenommen wurde.
- `PollingErrorCount`— Die Anzahl der Fehler, bei denen Lambda Anfragen von der Ereignisquelle nicht abfragen konnte. Lambda gibt diese Metrikdaten nur aus, wenn ein Fehler aufgetreten ist.

- `InvokeErrorCount`— Die Anzahl der Fehler, bei denen Lambda Ihre Event-Funktion nicht aufrufen konnte. Beachten Sie, dass es sich bei dem Aufruf um Datensätze im Batch handelt. Die Nummer steht auf Batch-Ebene, nicht auf Datensatzzählebene. Lambda gibt diese Metrikdaten nur aus, wenn ein Fehler aufgetreten ist.
- `SchemaRegistryErrorCount`— Die Anzahl der Fehler, bei denen Lambda das Schema nicht abrufen oder mit dem Schema deserialisieren konnte. Lambda gibt diese Metrikdaten nur aus, wenn ein Fehler aufgetreten ist.
- `CommitErrorCount`— Die Anzahl der Fehler, die Lambda nicht an den Kafka-Cluster übertragen konnte. Lambda gibt diese Metrikdaten nur aus, wenn ein Fehler aufgetreten ist.
- `MaxOffsetLag`— Die maximale Anzahl von Offset-Lags (Unterschied zwischen den letzten und den festgeschriebenen Offsets) für alle Partitionen in der Ereignisquellenzuordnung.
- `SumOffsetLag`— Die Summe der Offset-Lags auf allen Partitionen in der Ereignisquellenzuordnung.

Wenn Ihre Zuordnung von Ereignisquellen deaktiviert ist, erhalten Sie keine Metriken zur Zuordnung von Ereignisquellen. Möglicherweise werden Ihnen auch fehlende Metriken angezeigt, wenn CloudWatch die Verfügbarkeit von Lambda beeinträchtigt ist.

Arbeiten mit Lambda Event Function-Logs

Überwacht Lambda Event Functions AWS Lambda automatisch in Ihrem Namen, um Ihnen bei der Behebung von Fehlern zu helfen. Sie können Protokolle für Lambda-Ereignisfunktionen mithilfe der Lambda-Konsole, der CloudWatch Konsole, der AWS Command Line Interface (AWS CLI) und der API anzeigen. CloudWatch Sie können Lambda auch so konfigurieren, dass Protokolle an Amazon S3 und Firehose gesendet werden.

Solange die Ausführungsrolle Ihrer Event Function über die erforderlichen Berechtigungen verfügt, erfasst Lambda Logs für alle Anfragen, die von Ihrer Event Function bearbeitet werden, und sendet sie an Amazon CloudWatch Logs, das Standardziel. Sie können die Lambda-Konsole auch verwenden, um Amazon S3 oder Firehose als Protokollierungsziele zu konfigurieren.

- CloudWatch Logs ist das Standardprotokollierungsziel für Lambda Event Functions. CloudWatch Logs bietet Funktionen zur Anzeige und Analyse von Protokollen in Echtzeit und unterstützt die Erstellung von Metriken und Alarmen auf der Grundlage Ihrer Protokolldaten.
- Amazon S3 ist wirtschaftlich für die langfristige Speicherung, und Services wie Athena können zur Analyse von Protokollen verwendet werden. Die Latenz ist in der Regel höher.
- Firehose bietet verwaltetes Streaming von Protokollen an verschiedene Ziele. Wenn Sie Protokolle an andere AWS Dienste (z. B. OpenSearch Service oder Redshift Data API) oder Plattformen von Drittanbietern (wie Datadog, New Relic oder Splunk) senden müssen, vereinfacht Firehose diesen Prozess durch die Bereitstellung vorgefertigter Integrationen. Sie können auch an benutzerdefinierte HTTP-Endpunkte streamen, ohne zusätzliche Infrastruktur einrichten zu müssen.

Auswahl eines Serviceziels für das Senden von Protokollen

Berücksichtigen Sie bei der Auswahl eines Dienstes als Ziel für Event Function Logs die folgenden Schlüsselfaktoren:

- Das Kostenmanagement ist je nach Service unterschiedlich. Amazon S3 bietet in der Regel die wirtschaftlichste Option für die Langzeitspeicherung, während CloudWatch Logs die Möglichkeit bietet, Protokolle anzuzeigen, zu verarbeiten und Benachrichtigungen in Echtzeit einzurichten. Die Kosten für Firehose umfassen sowohl den Streamingservice als auch die Kosten für die von Ihnen konfigurierten Streamingziele.
- Die Analysefunktionen unterscheiden sich je nach Service. CloudWatch Logs zeichnet sich durch eine hervorragende Echtzeitüberwachung aus und lässt sich nativ in andere CloudWatch

Funktionen wie Logs Insights und Live Tail integrieren. Amazon S3 funktioniert gut mit Analysetools wie Athena und kann in verschiedene Dienste integriert werden, obwohl dafür möglicherweise eine zusätzliche Einrichtung erforderlich ist. Firehose vereinfacht das direkte Streaming zu bestimmten AWS Diensten (wie OpenSearch Service und Redshift Data API) und unterstützten Plattformen von Drittanbietern (wie Datadog und Splunk), indem vorgefertigte Integrationen bereitgestellt werden, wodurch der Konfigurationsaufwand potenziell reduziert wird.

- Einrichtung und Benutzerfreundlichkeit variieren je nach Service. CloudWatch Logs ist das standardmäßige Protokollziel. Es funktioniert sofort ohne zusätzliche Konfiguration und ermöglicht die einfache Anzeige und Analyse von Protokollen über die CloudWatch Konsole. Wenn Sie Protokolle an Amazon S3 senden möchten, müssen Sie zunächst einige Einstellungen in der Lambda-Konsole vornehmen und Bucket-Berechtigungen konfigurieren. Wenn Sie Protokolle benötigen, die direkt an Dienste wie OpenSearch Service oder Analyseplattformen von Drittanbietern gesendet werden, kann Firehose diesen Prozess vereinfachen.

Konfigurieren von Protokollzielen

AWS Lambda unterstützt mehrere Ziele für Ihre Event Function-Logs. In diesem Leitfaden werden die verfügbaren Protokollziele erläutert und Ihnen bei der Auswahl der für Ihre Anforderungen geeigneten Option geholfen. Unabhängig vom gewählten Ziel bietet Lambda Optionen zur Steuerung des Protokollformats, der Filterung und der Zustellung.

Lambda unterstützt sowohl JSON- als auch Klartext-Formate für die Protokolle Ihrer Event Function. Strukturierte JSON-Protokolle bieten eine verbesserte Suchbarkeit und ermöglichen eine automatisierte Analyse, während Klartextprotokolle einfach zu handhaben sind und potenziell geringere Speicherkosten verursachen. Sie können steuern, welche Protokolle Lambda an das von Ihnen gewählte Ziel sendet, indem Sie Protokollebenen für System- und Anwendungsprotokolle konfigurieren. Die Filterung hilft Ihnen, die Speicherkosten zu verwalten, und erleichtert das Auffinden relevanter Protokolleinträge beim Debuggen.

Ausführliche Anweisungen zur Einrichtung für jedes Ziel finden Sie in den folgenden Abschnitten:

- Lambda Event Function-Logs an CloudWatch Logs senden
- Lambda Event Function-Protokolle an Firehose senden
- Lambda Event Function-Protokolle an Amazon S3 senden

Konfiguration erweiterter Protokollierungssteuerelemente für Lambda-Ereignisfunktionen

Um Ihnen mehr Kontrolle darüber zu geben, wie Ihre Event Function-Logs erfasst, verarbeitet und genutzt werden, bietet Lambda die folgenden Konfigurationsoptionen für die Protokollierung:

- Protokollformat — wählen Sie zwischen Klartext und strukturiertem JSON-Format für die Protokolle Ihrer Event Function.
- Protokollebene — Wählen Sie für strukturierte JSON-Logs die Detailebene der Logs aus, an die Lambda sendet CloudWatch`FATAL`, z. B.`ERROR`,`WARN`,`INFO`,`DEBUG`, und`TRACE`.
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Event Function Logs sendet.

Weitere Informationen zur Konfiguration erweiterter Protokollierungssteuerungen finden Sie in den folgenden Abschnitten:

- Konfiguration der JSON- und Klartext-Protokollformate
- Log-level filtern
- Konfiguration von CloudWatch Protokollgruppen

Konfiguration der JSON- und Klartext-Protokollformate

Das Erfassen Ihrer Protokollausgaben als JSON-Schlüsselwertpaare erleichtert das Suchen und Filtern beim Debuggen Ihrer Event-Funktionen. Mit Protokollen im JSON-Format können Sie Ihren Protokollen auch Tags und Kontextinformationen hinzufügen. Das kann Ihnen bei der automatisierten Analyse großer Mengen von Protokolldaten helfen. Sofern Ihr Entwicklungsworkflow nicht auf vorhandenen Tools basiert, die Lambda-Protokolle im Klartext verarbeiten, empfehlen wir Ihnen, JSON als Protokollformat auszuwählen.

> **Von Lambda verwaltete Instances**
>
> Lambda Managed Instances unterstützen nur das JSON-Protokollformat. Wenn Sie eine Funktion für verwaltete Instanzen erstellen, konfiguriert Lambda das Protokollformat automatisch auf JSON, und Sie können es nicht in Klartext ändern. Weitere Informationen zu Managed Instances finden Sie unter. the section called “Verwaltete Instances”

Für alle von Lambda verwalteten Laufzeiten können Sie wählen, ob die Systemprotokolle Ihrer Event Function im unstrukturierten Klartext- oder JSON-Format an CloudWatch Logs gesendet werden. Systemprotokolle sind die von Lambda generierten Protokolle und werden manchmal auch als Plattformereignisprotokolle bezeichnet.

Wenn Sie eine der unterstützten integrierten Protokollierungsmethoden verwenden, kann Lambda für unterstützte Laufzeiten auch die Anwendungsprotokolle Ihrer Event-Funktion (die Protokolle, die Ihr Event-Funktionscode generiert) im strukturierten JSON-Format ausgeben. Wenn Sie das Protokollformat Ihrer Event-Funktion für diese Laufzeiten konfigurieren, gilt die von Ihnen gewählte Konfiguration sowohl für System- als auch für Anwendungsprotokolle.

Wenn Ihre Event-Funktion für unterstützte Laufzeiten eine unterstützte Protokollierungsbibliothek oder -methode verwendet, müssen Sie keine Änderungen an Ihrem vorhandenen Code vornehmen, damit Lambda Logs in strukturiertem JSON erfasst.

> **Note**
>
> Durch die Verwendung der JSON-Protokollformatierung werden zusätzliche Metadaten hinzugefügt und Protokollmeldungen als JSON-Objekte kodiert, die eine Reihe von Schlüssel-Wert-Paaren enthalten. Aus diesem Grund kann die Größe der Protokollnachrichten Ihrer Event-Funktion zunehmen.

Unterstützte Laufzeiten und Protokollierungsmethoden

Lambda unterstützt derzeit die Option, strukturierte JSON-Anwendungsprotokolle für folgende Laufzeiten auszugeben.

Sprache	Unterstützte Versionen
Java	Alle Java-Laufzeiten außer Java 8 auf Amazon Linux 1
.NET	.NET 8 und höher
Node.js	Node.js 16 und später
Python	Python 3.8 oder höher
Ruby	Ruby 4.0 und höher

Sprache	Unterstützte Versionen
Rust	n/a

Damit Lambda die Anwendungsprotokolle Ihrer Event Function CloudWatch im strukturierten JSON-Format senden kann, muss Ihre Event Function die folgenden integrierten Protokollierungstools verwenden, um Logs auszugeben:

- Java: Der `LambdaLogger` Logger oder Log4j2. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- .NET: Die `ILambdaLogger` Instanz auf dem Kontextobjekt. Weitere Informationen finden Sie unter Protokollieren und Überwachen von C#-Lambda-Ereignisfunktionen.
- Node.js- Die Konsolenmethoden `console.trace`,`console.debug`,`console.log`,`console.info`,`console.error`, und`console.warn`. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- Python: Die `logging` Standard-Python-Bibliothek. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- Ruby: Die `Logger` Standard-Ruby-Bibliothek. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- Rust: Die `tracing` Kiste. Weitere Informationen finden Sie unter the section called “Protokollierung”.

Für andere verwaltete Lambda-Laufzeiten unterstützt Lambda derzeit nur nativ die Erfassung von Systemprotokollen im strukturierten JSON-Format. Sie können jedoch weiterhin Anwendungsprotokolle zu jeder Laufzeit im strukturierten JSON-Format erfassen, indem Sie Protokollierungstools wie Powertools für AWS Lambda die Ausgabe von Protokollausgaben im JSON-Format verwenden.

Standardprotokollformate

Derzeit ist das Standard-Protokollformat für alle Lambda-Laufzeiten das Klartextformat. Für Lambda Managed Instances ist das Protokollformat immer JSON und kann nicht geändert werden.

Wenn Sie bereits Logging-Bibliotheken wie Powertools verwenden, AWS Lambda um Ihre Event Function-Logs im strukturierten JSON-Format zu generieren, müssen Sie Ihren Code nicht ändern, wenn Sie die JSON-Protokollformatierung auswählen. Lambda codiert keine Protokolle doppelt, die

bereits JSON-kodiert sind, sodass die Anwendungsprotokolle Ihrer Event Function weiterhin wie zuvor erfasst werden.

JSON-Format für Systemprotokolle

Wenn Sie das Protokollformat Ihrer Event-Funktion als JSON konfigurieren, wird jedes Systemprotokollelement (Plattformereignis) als JSON-Objekt erfasst, das Schlüsselwertpaare mit den folgenden Schlüsseln enthält:

- `"time"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"type"` – die Art des Ereignisses, das protokolliert wird
- `"record"` – der Inhalt der Protokollausgabe

Das Format des `"record"`-Werts hängt von der Art des protokollierten Ereignisses ab. Weitere Informationen finden Sie unter the section called “Objekttypen für Telemetrie-API-Ereignisse”. Weitere Hinweise zu den Protokollebenen, die Systemprotokollereignissen zugewiesen sind, finden Sie unter the section called “Zuordnung von Ereignissen auf Systemprotokollebene”.

Zum Vergleich zeigen die folgenden beiden Beispiele dieselbe Protokollausgabe sowohl im Klartext- als auch im strukturierten JSON-Format. Beachten Sie, dass Systemprotokollereignisse in den meisten Fällen mehr Informationen enthalten, wenn sie im JSON-Format ausgegeben werden, als wenn sie im Klartext ausgegeben werden.

Example Klartext:

```
2024-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.12.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example strukturiertes JSON:

```
{
  "time": "2024-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.12.v18",
```

```
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
  }
}
```

> **Note**
>
> Die the section called “Telemetrie-API” gibt immer Plattformereignisse wie `START` und `REPORT` im JSON-Format aus. Die Konfiguration des Formats der Systemprotokolle, an die Lambda sendet, CloudWatch hat keinen Einfluss auf das Verhalten der Lambda-Telemetrie-API.

JSON-Format für Anwendungsprotokolle

Wenn Sie das Protokollformat Ihrer Event-Funktion als JSON konfigurieren, werden Anwendungsprotokollausgaben, die mit unterstützten Protokollierungsbibliotheken und -methoden geschrieben wurden, als JSON-Objekt erfasst, das Schlüsselwertpaare mit den folgenden Schlüsseln enthält.

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"`(Python, .NET und Node.js) oder `"AWSrequestId"` (Java) — die eindeutige Anforderungs-ID für den Aufruf der Event-Funktion

Je nach Laufzeit und Protokollierungsmethode, die Ihre Funktion verwendet, kann dieses JSON-Objekt auch zusätzliche Schlüsselpaare enthalten. Wenn Ihre Funktion beispielsweise `console` Methoden verwendet Node.js, um Fehlerobjekte mit mehreren Argumenten zu protokollieren, enthält das JSON-Objekt zusätzliche Schlüssel-Wert-Paare mit den Schlüsseln `errorMessageerrorType`, und`stackTrace`. Weitere Informationen zu JSON-formatierten Protokollen in verschiedenen Lambda-Laufzeiten finden Sie unter the section called “Protokollierung”, the section called “Protokollierung” und the section called “Protokollierung”.

> **Note**
>
> Der Schlüssel, den Lambda für den Zeitstempelwert verwendet, unterscheidet sich für Systemprotokolle und für Anwendungsprotokolle. Bei Systemprotokollen verwendet Lambda

den Schlüssel `"time"`, um die Konsistenz mit der Telemetrie-API aufrechtzuerhalten. Bei Anwendungsprotokollen folgt Lambda den Konventionen der unterstützten Laufzeiten und verwendet `"timestamp"`.

Zum Vergleich zeigen die folgenden beiden Beispiele dieselbe Protokollausgabe sowohl im Klartext- als auch im strukturierten JSON-Format.

Example Klartext:

```
2024-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example strukturiertes JSON:

```
{
    "timestamp":"2024-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Das Protokollformat Ihrer Event-Funktion festlegen

Um das Protokollformat für Ihre Event-Funktion zu konfigurieren, können Sie die Lambda-Konsole oder die AWS Command Line Interface (AWS CLI) verwenden. Sie können das Protokollformat einer Event-Funktion auch mithilfe der API-Befehle CreateFunctionund UpdateFunctionConfigurationLambda, der Ressource AWS Serverless Application Model (AWS SAM) AWS::Serverless: :Function und der Ressource CloudFormation AWS::Lambda: :Function konfigurieren.

Das Ändern des Protokollformats Ihrer Event-Funktion hat keine Auswirkungen auf bestehende Protokolle, die in Logs gespeichert sind. CloudWatch Nur neue Protokolle verwenden das aktualisierte Format.

Wenn Sie das Protokollformat Ihrer Event-Funktion in JSON ändern und keine Protokollebene festlegen, setzt Lambda die Anwendungsprotokollebene und die Systemprotokollebene Ihrer Event-Funktion automatisch auf INFO. Das bedeutet, dass Lambda nur Protokollausgaben der Stufe INFO und niedriger an CloudWatch Logs sendet. Weitere Informationen zur Filterung auf Anwendungs- und Systemprotokollebene finden Sie unter the section called “Log-level filtern”

> **Note**
>
> Wenn für Python-Laufzeiten das Protokollformat Ihrer Event-Funktion auf Klartext gesetzt ist, ist die Standardeinstellung auf Protokollebene WARN. Das bedeutet, dass Lambda nur Protokollausgaben der Stufe WARN und niedriger an CloudWatch Logs sendet. Wenn Sie das Protokollformat Ihrer Event-Funktion in JSON ändern, ändert sich dieses Standardverhalten. Weitere Informationen zur Protokollierung in Python finden Sie unter the section called “Protokollierung”.

Bei Node.js Funktionen, die EMF-Logs (Embedded Metric Format) ausgeben, kann die Änderung des Protokollformats Ihrer Event-Funktion in JSON dazu führen, dass Ihre Metriken nicht erkannt CloudWatch werden können.

> **Important**
>
> Wenn Ihre Event-Funktion Powertools for AWS Lambda (TypeScript) oder die Open-Source-EMF-Client-Bibliotheken zur Ausgabe von EMF-Protokollen verwendet, aktualisieren Sie Ihre Powertools - und EMF-Bibliotheken auf die neuesten Versionen, um sicherzustellen, dass Ihre Protokolle weiterhin korrekt analysiert CloudWatch werden können. Wenn Sie zum JSON-Protokollformat wechseln, empfehlen wir Ihnen außerdem, Tests durchzuführen, um die Kompatibilität mit den eingebetteten Metriken Ihrer Event Function sicherzustellen. Weitere Hinweise zu den Funktionen von node.js, die EMF-Protokolle ausgeben, finden Sie unter the section called “Verwenden von Clientbibliotheken im Embedded Metric Format (EMF) mit strukturierten JSON-Protokollen”.

So konfigurieren Sie das Protokollformat einer Event-Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion
3. Wählen Sie auf der Konfigurationsseite für die Ereignisfunktion die Option Überwachungs- und Betriebstools aus.
4. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
5. Wählen Sie unter Protokollinhalt für Protokollformat entweder Text oder JSON aus.
6. Wählen Sie Speichern.

Um das Protokollformat einer vorhandenen Funktion zu ändern (AWS CLI)

- Verwenden Sie den Befehl update-function-configuration, um das Protokollformat einer vorhandenen Funktion zu ändern. Legen Sie die `LogFormat`-Option in `LoggingConfig` entweder auf `JSON` oder `Text` fest.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON
```

Um das Protokollformat festzulegen, wenn Sie eine Funktion erstellen (AWS CLI)

- Verwenden Sie die Option `--logging-config` im Befehl create-function, um das Protokollformat zu konfigurieren, wenn Sie eine neue Funktion erstellen. Legen Sie `LogFormat` auf `JSON` oder `Text` fest. Der folgende Beispielbefehl erstellt eine Node.js Funktion, die Logs in strukturiertem JSON ausgibt.

  Wenn Sie beim Erstellen einer Funktion kein Protokollformat angeben, verwendet Lambda das Standardprotokollformat für die von Ihnen ausgewählte Laufzeitversion. Informationen zu den Standard-Protokollierungsformaten finden Sie unter the section called “Standardprotokollformate”.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON
```

Log-level filtern

Lambda kann die Protokolle Ihrer Event-Funktion filtern, sodass nur Protokolle mit einer bestimmten Detailebene oder niedriger an CloudWatch Logs gesendet werden. Sie können die Filterung auf Protokollebene separat für die Systemprotokolle (die von Lambda generierten Protokolle) und Anwendungsprotokolle (die Protokolle, die Ihr Eventfunktionscode generiert) Ihrer Event Function konfigurieren.

Denn the section called “Unterstützte Laufzeiten und Protokollierungsmethoden” Sie müssen keine Änderungen an Ihrem Event-Funktionscode für Lambda vornehmen, um die Anwendungsprotokolle Ihrer Event-Funktion zu filtern.

Für alle anderen Laufzeiten und Protokollierungsmethoden muss Ihr Event-Funktionscode Log-Ereignisse in `stdout` oder `stderr` als JSON-formatierte Objekte ausgeben, die ein Schlüssel-Wert-Paar mit dem Schlüssel enthalten. `"level"` Lambda interpretiert beispielsweise die folgende Ausgabe an `stdout` als ein Protokoll auf Ebene DEBUG.

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2024-11-02T16:51:31.587199Z"}')
```

Wenn das `"level"` Wertfeld ungültig ist oder fehlt, weist Lambda der Protokollausgabe die Stufe INFO zu. Damit Lambda das Zeitstempelfeld verwenden kann, müssen Sie die Zeit im gültigen RFC 3339-Zeitstempelformat angeben. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Halten Sie sich bei der Benennung des Zeitstempelschlüssels an die Benennungskonventionen der Laufzeit, die Sie verwenden. Lambda unterstützt die meisten gängigen Namenskonventionen, die von den verwalteten Laufzeiten verwendet werden.

> **Note**
>
> Um die Filterung auf Protokollebene verwenden zu können, muss Ihre Event-Funktion so konfiguriert sein, dass sie das JSON-Protokollformat verwendet. Derzeit ist das Standard-Protokollformat für alle verwalteten Lambda-Laufzeiten das Klartextformat. Informationen zur Konfiguration des Protokollformats Ihrer Event-Funktion auf JSON finden Sie unter. the section called “Das Protokollformat Ihrer Event-Funktion festlegen”

Bei Anwendungsprotokollen (den durch Ihren Funktionscode generierten Protokollen) können Sie zwischen den folgenden Protokollebenen wählen.

Protokollebene	Standardnutzung
TRACE (am detailliertesten)	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen

Protokollebene	Standardnutzung
DEBUG	Detaillierte Informationen für das System-De bugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Meldungen über potenzielle Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert
FATAL (am wenigsten Details)	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Wenn Sie eine Protokollebene auswählen, sendet Lambda Protokolle auf dieser Ebene und niedriger an CloudWatch Logs. Wenn Sie beispielsweise die Anwendungsprotokollebene einer Ereignisfunktion auf WARN setzen, sendet Lambda keine Protokollausgaben auf den Ebenen INFO und DEBUG. Die Standardebene des Anwendungsprotokolls für die Protokollfilterung ist INFO.

Wenn Lambda die Anwendungsprotokolle Ihrer Event Function filtert, wird Protokollnachrichten ohne Stufe die Protokollebene INFO zugewiesen.

Für Systemprotokolle (die vom Lambda-Service generierten Protokolle) können Sie zwischen den folgenden Protokollebenen wählen.

Protokollebene	Usage
DEBUG (am detailliertesten)	Detaillierte Informationen für das System-De bugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen

Protokollebene	Usage
WARN (am wenigsten Details)	Meldungen über potenzielle Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden

Wenn Sie eine Protokollebene auswählen, sendet Lambda Protokolle auf dieser Ebene und niedriger. Wenn Sie beispielsweise die Systemprotokollebene einer Event-Funktion auf INFO setzen, sendet Lambda keine Protokollausgaben auf der DEBUG-Ebene.

Standardmäßig setzt Lambda die Protokollebene des Systems auf INFO. Mit dieser Einstellung sendet `"start"` und `"report"` protokolliert Lambda automatisch Nachrichten an CloudWatch. Um mehr oder weniger detaillierte Systemprotokolle zu erhalten, ändern Sie die Protokollebene in DEBUG oder WARN. Eine Liste der Protokollebenen, denen Lambda verschiedene Systemprotokollereignisse zuordnet, finden Sie unter the section called “Zuordnung von Ereignissen auf Systemprotokollebene”.

Konfigurieren der Filterung auf Protokollebene

Um die Filterung auf Anwendungs- und Systemprotokollebene für Ihre Event Function zu konfigurieren, können Sie die Lambda-Konsole oder die AWS Command Line Interface () verwenden.AWS CLI Sie können die Protokollebene einer Event-Funktion auch mithilfe der API-Befehle CreateFunctionund UpdateFunctionConfigurationLambda, der Ressource AWS Serverless Application Model (AWS SAM) AWS::Serverless: :Function und der Ressource CloudFormation AWS::Lambda: :Function konfigurieren.

Beachten Sie, dass, wenn Sie die Protokollebene Ihrer Ereignisfunktion in Ihrem Code festlegen, diese Einstellung Vorrang vor allen anderen von Ihnen konfigurierten Einstellungen auf Protokollebene hat. Wenn Sie beispielsweise die `logging` `setLevel()` Python-Methode verwenden, um die Protokollierungsebene Ihrer Event-Funktion auf INFO zu setzen, hat diese Einstellung Vorrang vor der Einstellung WARN, die Sie mit der Lambda-Konsole konfigurieren.

Um die Anwendungs- oder Systemprotokollebene (Konsole) einer vorhandenen Event-Funktion zu konfigurieren

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.

3. Wählen Sie auf der Konfigurationsseite für die Ereignisfunktion die Option Überwachungs- und Betriebstools aus.
4. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
5. Stellen Sie sicher, dass unter Protokollinhalt für Protokollformat JSON ausgewählt ist.
6. Wählen Sie mithilfe der Optionsfelder die gewünschte Anwendungsprotokollebene und die gewünschte Systemprotokollebene für Ihre Ereignisfunktion aus.
7. Wählen Sie Speichern.

Um die Anwendungs- oder Systemprotokollebene einer vorhandenen Funktion zu konfigurieren (AWS CLI)

- Verwenden Sie den Befehl update-function-configuration, um die Anwendungs- oder Systemprotokollebene einer vorhandenen Funktion zu ändern. Verwenden Sie `--logging-config`, um `SystemLogLevel` auf einen der Werte `DEBUG`, `INFO` oder `WARN` zu setzen. Legen Sie `ApplicationLogLevel` auf `DEBUG`, `INFO`, `WARN`, `ERROR` oder `FATAL` fest.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

So konfigurieren Sie die Filterung auf Protokollebene beim Erstellen einer Funktion

- Um die Filterung der Protokollebene zu konfigurieren, wenn Sie eine neue Funktion erstellen, verwenden Sie `--logging-config`, um die Tasten `SystemLogLevel` und `ApplicationLogLevel` im Befehl create-function zu setzen. Legen Sie `SystemLogLevel` auf `DEBUG`, `INFO` oder `WARN` fest. Legen Sie `ApplicationLogLevel` auf `DEBUG`, `INFO`, `WARN`, `ERROR` oder `FATAL` fest.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

Zuordnung von Ereignissen auf Systemprotokollebene

Für von Lambda generierte Protokollereignisse auf Systemebene ist in der folgenden Tabelle die Protokollebene definiert, die jedem Ereignis zugewiesen ist. Weitere Informationen zu den in der Tabelle aufgeführten Ereignissen finden Sie unter the section called “Referenz zum `Event`-Schema”

Ereignisname	Bedingung	Zugewiesene Protokollebene
initStart	„runtimeVersion“ ist festgelegt	INFO
initStart	„runtimeVersion“ ist nicht festgelegt	DEBUG
init RuntimeDone	status=success	DEBUG
init RuntimeDone	status!=success	WARN
initReport	initializationType!=on-demand	INFO
initReport	initializationType=on-demand	DEBUG
initReport	status!=success	WARN
restoreStart	„runtimeVersion“ ist festgelegt	INFO
restoreStart	„runtimeVersion“ ist nicht festgelegt	DEBUG
wiederherstellen RuntimeDone	status=success	DEBUG
wiederherstellen RuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
start	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN

Ereignisname	Bedingung	Zugewiesene Protokollebene
report	status=success	INFO
report	status!=success	WARN
extension	state=success	INFO
extension	state!=success	WARN
logSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

> **Note**
>
> Die the section called “Telemetrie-API” gibt immer den vollständigen Satz von Plattformereignissen aus. Die Konfiguration der Ebene der Systemprotokolle, an die Lambda sendet, CloudWatch hat keinen Einfluss auf das Verhalten der Lambda-Telemetrie-API.

Filterung auf Anwendungsprotokollebene mit benutzerdefinierten Laufzeiten

Wenn Sie die Filterung auf Anwendungsprotokollebene für Ihre Event Function konfigurieren, legt Lambda hinter den Kulissen die Anwendungsprotokollebene in der Laufzeit mithilfe der `AWS_LAMBDA_LOG_LEVEL` Umgebungsvariablen fest. Lambda legt auch das Protokollformat Ihrer Event-Funktion mithilfe der `AWS_LAMBDA_LOG_FORMAT` Umgebungsvariablen fest. Sie können diese Variablen verwenden, um erweiterte Lambda-Protokollierungsoptionen in eine benutzerdefinierte Laufzeit zu integrieren.

Um Protokollierungseinstellungen für eine Event-Funktion mithilfe einer benutzerdefinierten Laufzeit mit der Lambda-Konsole und Lambda-APIs konfigurieren zu können AWS CLI, konfigurieren Sie Ihre benutzerdefinierte Laufzeit so, dass sie den Wert dieser Umgebungsvariablen überprüft. Anschließend können Sie die Logger Ihrer Laufzeit gemäß dem von Ihnen ausgewählten Protokollformat und den von Ihnen ausgewählten Protokollebenen konfigurieren.

Lambda Event Function-Logs an CloudWatch Logs senden

Standardmäßig erfasst Lambda automatisch Protokolle für alle Aufrufe von Event Function und sendet sie an CloudWatch Logs, sofern die Ausführungsrolle Ihrer Event Function über die erforderlichen Berechtigungen verfügt. Diese Protokolle werden standardmäßig in einer Protokollgruppe mit dem Namen//gespeichert. aws/lambda *<function-name>* Um das Debugging zu verbessern, können Sie benutzerdefinierte Protokollierungsanweisungen in Ihren Code einfügen, die Lambda nahtlos in Logs integriert. CloudWatch Bei Bedarf können Sie Ihre Event-Funktion so konfigurieren, dass Protokolle mithilfe der Lambda-Konsole oder der Lambda-API an eine andere Gruppe gesendet werden. AWS CLI Weitere Informationen hierzu finden Sie unter the section called “ CloudWatch Funktionsprotokolle konfigurieren”.

Sie können Protokolle für Lambda-Ereignisfunktionen mithilfe der Lambda-Konsole, der CloudWatch Konsole, der AWS Command Line Interface (AWS CLI) oder der CloudWatch API anzeigen. Weitere Informationen finden Sie unter CloudWatch Protokolle für Lambda-Ereignisfunktionen anzeigen.

> **Note**
>
> Es kann 5 bis 10 Minuten dauern, bis die Protokolle nach dem Aufruf einer Event-Funktion angezeigt werden.

Erforderliche IAM-Berechtigungen

Ihre Ausführungsrolle benötigt die folgenden Berechtigungen, um Protokolle in Logs hochzuladen: CloudWatch

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

Weitere Informationen finden Sie unter Verwenden von identitätsbasierten Richtlinien (IAM-Richtlinien) für CloudWatch Logs im CloudWatch Amazon-Benutzerhandbuch.

Sie können diese CloudWatch Log-Berechtigungen mithilfe der von Lambda bereitgestellten `AWSLambdaBasicExecutionRole` AWS verwalteten Richtlinie hinzufügen. Führen Sie den folgenden Befehl aus, um diese Richtlinie zu Ihrer Rolle hinzuzufügen:

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

Weitere Informationen finden Sie unter the section called “AWS verwaltete Richtlinien”.

Preisgestaltung

Für die Verwendung von Lambda-Protokollen fallen keine zusätzlichen Gebühren an. Es fallen jedoch die Standardgebühren für CloudWatch Logs an. Weitere Informationen finden Sie unter CloudWatch Preise.

Konfiguration von CloudWatch Protokollgruppen

Standardmäßig wird CloudWatch automatisch eine Protokollgruppe erstellt, die `/aws/lambda/<function name>` nach Ihrer Lambda-Ereignisfunktion benannt ist, wenn sie zum ersten Mal aufgerufen wird. Um Ihre Event Function so zu konfigurieren, dass Logs an eine bestehende Log-Gruppe gesendet werden, oder um eine neue Loggruppe für Ihre Event Function zu erstellen, können Sie die Lambda-Konsole oder die AWS CLI verwenden. Sie können benutzerdefinierte Protokollgruppen auch mithilfe der Befehle CreateFunctionund UpdateFunctionConfigurationLambda API und der Ressource AWS Serverless Application Model (AWS SAM) AWS: :Serverless: :Function konfigurieren.

Sie können mehrere Lambda-Ereignisfunktionen konfigurieren, um Protokolle an dieselbe CloudWatch Protokollgruppe zu senden. Sie könnten beispielsweise eine einzelne Protokollgruppe verwenden, um Protokolle für alle Lambda-Ereignisfunktionen zu speichern, aus denen eine bestimmte Anwendung besteht. Wenn Sie eine benutzerdefinierte Protokollgruppe für eine Lambda-Ereignisfunktion verwenden, enthalten die von Lambda erstellten Protokollstreams den Namen der Ereignisfunktion und die Version der Ereignisfunktion. Dadurch wird sichergestellt, dass die Zuordnung zwischen Protokollnachrichten und Ereignisfunktionen beibehalten wird, auch wenn Sie dieselbe Protokollgruppe für mehrere Ereignisfunktionen verwenden.

Das Protokollstream-Benennungsformat für benutzerdefinierte Protokollgruppen folgt dieser Benennungskonvention:

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

Beachten Sie, dass bei der Konfiguration einer benutzerdefinierten Protokollgruppe der Name, den Sie für Ihre Protokollgruppe auswählen, den Benennungsregeln für CloudWatch Protokolle

entsprechen muss. Darüber hinaus dürfen benutzerdefinierte Protokollgruppennamen nicht mit der Zeichenfolge `aws/` beginnen. Wenn Sie eine benutzerdefinierte Protokollgruppe erstellen, die mit `aws/` beginnt, kann Lambda die Protokollgruppe nicht erstellen. Aus diesem Grund werden die Protokolle Ihrer Event-Funktion nicht an gesendet CloudWatch.

So ändern Sie die Protokollgruppe einer Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie auf der Konfigurationsseite für die Ereignisfunktion die Option Überwachungs- und Betriebstools aus.
4. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
5. Wählen Sie im Bereich Protokollgruppe für CloudWatch Protokollgruppe die Option Benutzerdefiniert aus.
6. Geben Sie unter Benutzerdefinierte Protokollgruppe den Namen der CloudWatch Protokollgruppe ein, an die Ihre Funktion Protokolle senden soll. Wenn Sie den Namen einer vorhandenen Protokollgruppe eingeben, verwendet Ihre Funktion diese Gruppe. Wenn keine Protokollgruppe mit dem von Ihnen eingegebenen Namen existiert, erstellt Lambda eine neue Protokollgruppe für Ihre Funktion mit diesem Namen.

Um die Protokollgruppe einer Ereignisfunktion zu ändern (AWS CLI)

- Um die Protokollgruppe einer vorhandenen Ereignisfunktion zu ändern, verwenden Sie den Befehl update-function-configuration.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogGroup=myLogGroup
```

Um eine benutzerdefinierte Protokollgruppe anzugeben, wenn Sie eine Ereignisfunktion erstellen (AWS CLI)

- Um eine benutzerdefinierte Protokollgruppe anzugeben, wenn Sie eine neue Lambda-Ereignisfunktion mit dem erstellen AWS CLI, verwenden Sie die `--logging-config` Option. Der folgende Beispielbefehl erstellt eine Node.js Lambda-Ereignisfunktion, die Protokolle an eine Protokollgruppe mit dem Namen `myLogGroup` sendet.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogGroup=myLogGroup
```

Berechtigungen für die Ausführungsrolle

Damit Ihre Event-Funktion Logs an CloudWatch Logs senden kann, muss sie über die PutLogEventsLogs-Berechtigung verfügen. Wenn Sie die Protokollgruppe Ihrer Event-Funktion mithilfe der Lambda-Konsole konfigurieren, fügt Lambda der Rolle diese Berechtigung unter den folgenden Bedingungen hinzu:

- Das Serviceziel ist auf Logs festgelegt CloudWatch
- Die Ausführungsrolle Ihrer Event-Funktion ist nicht berechtigt, Logs in Logs (das Standardziel) hochzuladen CloudWatch

> **Note**
>
> Lambda fügt keine Put-Berechtigungen für Amazon S3- oder Firehose-Protokollziele hinzu.

Wenn Lambda diese Berechtigung hinzufügt, erteilt es der Event Function die Berechtigung, Protokolle an jede CloudWatch Logs-Protokollgruppe zu senden.

Um zu verhindern, dass Lambda die Ausführungsrolle der Event-Funktion automatisch aktualisiert und sie stattdessen manuell bearbeitet, erweitern Sie Permissions und deaktivieren Sie Add required permissions.

Wenn Sie die Protokollgruppe Ihrer Event-Funktion mithilfe von konfigurieren AWS CLI, fügt Lambda die `logs:PutLogEvents` Berechtigung nicht automatisch hinzu. Fügen Sie die Berechtigung zur Ausführungsrolle Ihrer Event-Funktion hinzu, falls sie noch nicht vorhanden ist. Diese Berechtigung ist in der AWSLambdaBasicExecutionRoleverwalteten Richtlinie enthalten.

CloudWatch Protokollierung für Lambda Managed Instances

Bei der Verwendung von Lambda Managed Instances gibt es zusätzliche Überlegungen zum Senden von Protokollen an CloudWatch Logs:

VPC-Netzwerkanforderungen

Lambda Managed Instances werden auf kundeneigenen EC2-Instances innerhalb Ihrer VPC ausgeführt. Um Logs an CloudWatch Logs und Traces zu senden X-Ray, müssen Sie sicherstellen, dass diese AWS APIs von Ihrer VPC aus routingfähig sind. Sie haben mehrere Möglichkeiten:

- AWS PrivateLink (empfohlen): Wird verwendet AWS PrivateLink, um VPC-Endpunkte für CloudWatch Protokolle und X-Ray Dienste zu erstellen. Dadurch können Ihre Instances privat auf diese Dienste zugreifen, ohne dass ein Internet-Gateway oder NAT-Gateway erforderlich ist. Weitere Informationen finden Sie unter Verwenden von CloudWatch Protokollen mit VPC-Endpunkten der Schnittstelle.
- NAT-Gateway: Konfigurieren Sie ein NAT-Gateway, um ausgehenden Internetzugang von Ihren privaten Subnetzen aus zu ermöglichen.
- Internet Gateway: Stellen Sie für öffentliche Subnetze sicher, dass für Ihre VPC ein Internet-Gateway konfiguriert ist.

Wenn CloudWatch Logs oder X-Ray APIs nicht von Ihrer VPC aus routingfähig sind, werden Ihre Event Function-Logs und -Traces nicht zugestellt.

Gleichzeitige Aufrufe und Protokollzuweisung

Ausführungsumgebungen für Lambda Managed Instances können mehrere Aufrufe gleichzeitig verarbeiten. Wenn mehrere Aufrufe gleichzeitig ausgeführt werden, werden ihre Protokolleinträge im selben Protokollstream verschachtelt. Um Protokolle von gleichzeitigen Aufrufen effektiv zu filtern und zu analysieren, sollten Sie sicherstellen, dass jeder Protokolleintrag die Anforderungs-ID enthält. AWS

Wir empfehlen einen der folgenden Ansätze:

- Verwenden Sie standardmäßige Lambda-Laufzeitlogger (empfohlen): Die von Lambda Managed Runtimes bereitgestellten Standard-Logging-Bibliotheken enthalten automatisch die Anforderungs-ID in jedem Protokolleintrag.
- Implementieren Sie strukturierte JSON-Protokollierung: Wenn Sie eine benutzerdefinierte Laufzeit erstellen oder eine benutzerdefinierte Protokollierung benötigen, implementieren Sie JSON-

formatted Protokolle, die die Anforderungs-ID in jedem Eintrag enthalten. Lambda Managed Instances unterstützen nur das JSON-Protokollformat. Nehmen Sie das `requestId` Feld in Ihre JSON-Logs auf, um die Filterung nach Aufrufen zu ermöglichen:

```
{
  "timestamp": "2025-01-15T10:30:00.000Z",
  "level": "INFO",
  "requestId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "message": "Processing request"
}
```

Mit der Zuordnung der Anfrage-ID können Sie CloudWatch Log-Log-Log-Einträge für einen bestimmten Aufruf mithilfe von CloudWatch Logs Insights-Abfragen filtern. Beispiel:

```
fields @timestamp, @message
| filter requestId = "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
| sort @timestamp asc
```

Weitere Informationen zu den Protokollierungsanforderungen für Lambda Managed Instances finden Sie unterGrundlegendes zur Lambda Managed Instance-Ausführungsumgebung.

CloudWatch Protokolle für Lambda-Ereignisfunktionen anzeigen

Sie können CloudWatch Amazon-Protokolle für Ihre Lambda-Ereignisfunktion mit der Lambda-Konsole, der CloudWatch Konsole oder der AWS Command Line Interface ()AWS CLI anzeigen. Folgen Sie den Anweisungen in den folgenden Abschnitten, um auf die Protokolle Ihrer Event Function zuzugreifen.

Streamen Sie die Protokolle der Event Function mit CloudWatch Logs Live Tail

Amazon CloudWatch Logs Live Tail hilft Ihnen bei der schnellen Fehlerbehebung bei Ihren Event-Funktionen, indem es eine Streaming-Liste mit neuen Protokollereignissen direkt in der Lambda-Konsole anzeigt. Sie können aufgenommene Logs aus Ihren Lambda Event Functions in Echtzeit anzeigen und filtern, sodass Sie Probleme schnell erkennen und lösen können.

> **Note**
>
> Bei Live Tail-Sitzungen fallen Kosten pro Minute an. Weitere Informationen zur Preisgestaltung finden Sie unter CloudWatch Amazon-Preise.

Vergleich von Live Tail und --log-type Tail

Es gibt mehrere Unterschiede zwischen CloudWatch Logs Live Tail und der Option LogType: Tail in der Lambda-API (`--log-type Tail`in der AWS CLI):

- `--log-type Tail` gibt nur die ersten 4 KB der Aufruf-Logs zurück. Live Tail hat dieses Limit nicht gemeinsam und kann bis zu 500 Protokollereignisse pro Sekunde empfangen.
- `--log-type Tail`erfasst und sendet die Protokolle zusammen mit der Antwort, was sich auf die Antwortlatenz der Event-Funktion auswirken kann. Live Tail hat keinen Einfluss auf die Reaktionslatenz der Event-Funktion.
- `--log-type Tail` unterstützt nur synchrone Aufrufe. Live Tail funktioniert sowohl für synchrone als auch für asynchrone Aufrufe.

> **Note**
>
> Lambda Managed Instances unterstützt `--log-type Tail` diese Option nicht. Verwenden Sie CloudWatch Logs Live Tail oder fragen Sie CloudWatch Logs direkt ab, um Logs für Funktionen von Managed Instances einzusehen.

Berechtigungen

Die folgenden Berechtigungen sind erforderlich, um CloudWatch Logs Live Tail-Sitzungen zu starten und zu beenden:

- `logs:DescribeLogGroups`
- `logs:StartLiveTail`
- `logs:StopLiveTail`

Starten einer Live-Tail-Sitzung in der Lambda-Konsole

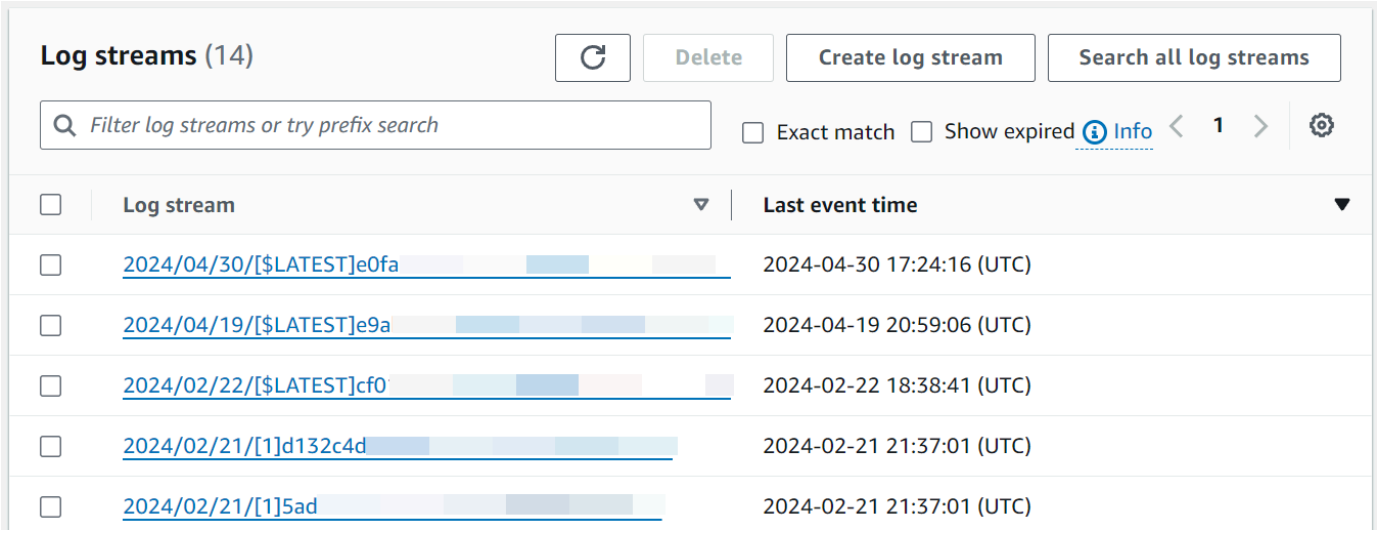
1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Event-Funktion.
3. Wählen Sie die Registerkarte Test.
4. Wählen Sie im Bereich Testereignis die Option CloudWatch Logs Live Tail aus.

5. Für „Protokollgruppen auswählen“ ist standardmäßig die Protokollgruppe der Ereignisfunktion ausgewählt. Sie können bis zu fünf Protokollgruppen gleichzeitig auswählen.
6. (Optional) Um nur Protokollereignisse anzuzeigen, die bestimmte Wörter oder andere Zeichenfolgen enthalten, geben Sie das Wort oder die Zeichenfolge in das Feld Filtermuster hinzufügen ein. Im Feld „Filter“ wird zwischen Groß- und Kleinschreibung unterschieden. Sie können mehrere Begriffe und Musteroperatoren in dieses Feld eingeben, einschließlich regulärer Ausdrücke (regex). Weitere Informationen zur Mustersyntax finden Sie unter Filtermustersyntax. im Amazon CloudWatch Logs-Benutzerhandbuch.
7. Wählen Sie Starten. Im Fenster werden nun übereinstimmende Protokollereignisse angezeigt.
8. Zum Beenden der Live-Tail-Sitzung wählen Sie Stoppen.

> **Note**
>
> Die Live-Tail-Sitzung wird automatisch nach 15 Minuten Inaktivität oder nach Ablauf der Zeit für die Lambda-Konsolensitzung beendet.

Greifen Sie über die Konsole auf die Event-Funktionsprotokolle zu

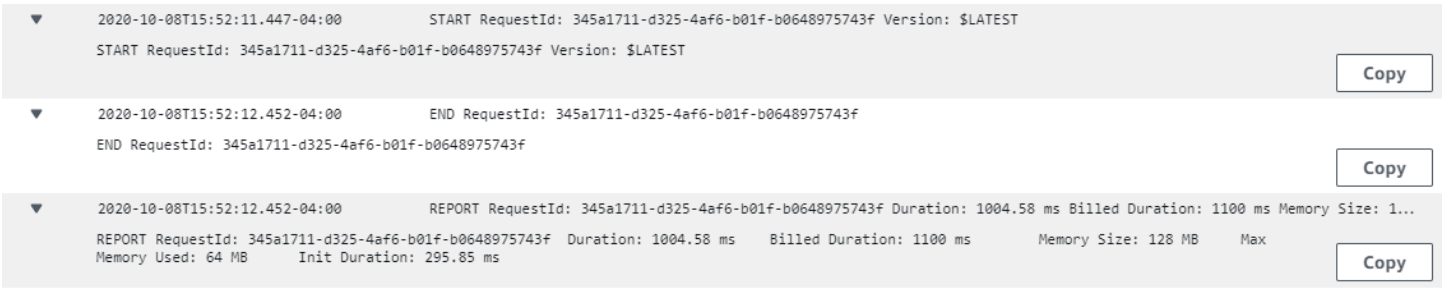
1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Ereignisfunktion aus.
3. Wählen Sie den Tab Überwachung.
4. Wählen Sie CloudWatch Protokolle anzeigen, um die CloudWatch Konsole zu öffnen.
5. Scrollen Sie nach unten und wählen Sie den Protokollstream für die Funktionsaufrufen aus, die Sie sich ansehen möchten.



Jede Instanz einer Lambda-Ereignisfunktion hat einen eigenen Protokollstream. Wenn eine Event-Funktion hochskaliert wird, hat jede gleichzeitige Instanz ihren eigenen Log-Stream. Jedes Mal, wenn als Reaktion auf einen Aufruf eine neue Ausführungsumgebung erstellt wird, wird ein neuer Protokollstream generiert. Die Benennungskonvention für Protokollstreams lautet:

```
YYYY/MM/DD[Function version][Execution environment GUID]
```

Eine einzelne Ausführungsumgebung schreibt während ihrer Lebensdauer in denselben Protokollstream. Der Protokollstream enthält Nachrichten aus dieser Ausführungsumgebung sowie alle Ausgaben aus dem Code Ihrer Lambda-Ereignisfunktion. Jede Nachricht wird mit einem Zeitstempel versehen, auch Ihre benutzerdefinierten Protokolle. Auch wenn Ihre Event-Funktion keine Ausgabe Ihres Codes protokolliert, werden pro Aufruf mindestens drei Protokollanweisungen generiert (START, END und REPORT):



Aus diesen Protokollen geht Folgendes hervor:

- RequestId— Dies ist eine eindeutige ID, die pro Anfrage generiert wird. Wenn die Lambda-Ereignisfunktion eine Anfrage erneut versucht, ändert sich diese ID nicht und erscheint in den Protokollen für jeden weiteren Versuch.
- Start/End— Diese setzen ein Lesezeichen für einen einzelnen Aufruf, sodass jede Protokollzeile zwischen diesen beiden zu demselben Aufruf gehört.
- Dauer: Die gesamte Aufrufzeit für die Handler-Funktion, ohne `INIT`-Code.
- Dauer der Abrechnung: Wendet die Rundungslogik für Abrechnungszwecke an.
- Speichergröße — die Menge an Speicher, die der Event-Funktion zugewiesen ist.
- Maximal genutzter Speicher: Der maximale während des Aufrufs genutzte Speicher.
- Initialisierungsdauer: Die Zeit, die benötigt wurde, um den `INIT`-Codeabschnitt außerhalb des Haupthandlers auszuführen.

Zugriffs-Logs mit dem AWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-`

`in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS und Linux (nur diese Systeme)

In derselben Befehlszeile müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Parsing von Protokollen und strukturierte Protokollierung

Mit CloudWatch Logs Insights können Sie Protokolldaten mithilfe einer speziellen Abfragesyntax suchen und analysieren. Es führt Abfragen über mehrere Protokollgruppen durch und bietet eine leistungsstarke Filterung mithilfe von Mustervergleichen mit Glob- und regulären Ausdrücken.

Sie können diese Funktionen nutzen, indem Sie eine strukturierte Protokollierung in Ihren Lambda-Ereignisfunktionen implementieren. Strukturierte Protokollierung organisiert Ihre Protokolle in einem vordefinierten Format, wodurch die Abfrage vereinfacht wird. Die Verwendung von Protokollebenen ist ein wichtiger erster Schritt zur Erstellung filterfreundlicher Protokolle, die Informationsmeldungen von Warnungen oder Fehlern trennen. Betrachten Sie zum Beispiel den folgenden Node.js Code:

```
exports.handler = async (event) => {
    console.log("console.log - Application is fine")
```

```
    console.info("console.info - This is the same as console.log")
    console.warn("console.warn - Application provides a warning")
    console.error("console.error - An error occurred")
}
```

Die resultierende CloudWatch Protokolldatei enthält ein separates Feld, das die Protokollebene angibt:

```
START RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109 Version: $LATEST
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.log - Application is fine
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.info - This is the same as console.log
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    WARN    console.warn - Application provides a warning
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    ERROR   console.error - An error occurred
END RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109
REPORT RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109  Duration: 3.13 ms       Billed Duration: 100 ms Memory Size: 128 MB
Max Memory Used: 64 MB  Init Duration: 142.18 ms
```

Eine CloudWatch Logs Insights-Abfrage kann dann auf Protokollebene filtern. Um beispielsweise nur nach Fehlern zu suchen, können Sie die folgende Abfrage verwenden:

```
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
```

JSON-strukturierte Protokollierung

JSON wird häufig verwendet, um Anwendungsprotokolle zu strukturieren. Im folgenden Beispiel wurden die Protokolle in JSON konvertiert, um drei unterschiedliche Werte auszugeben:

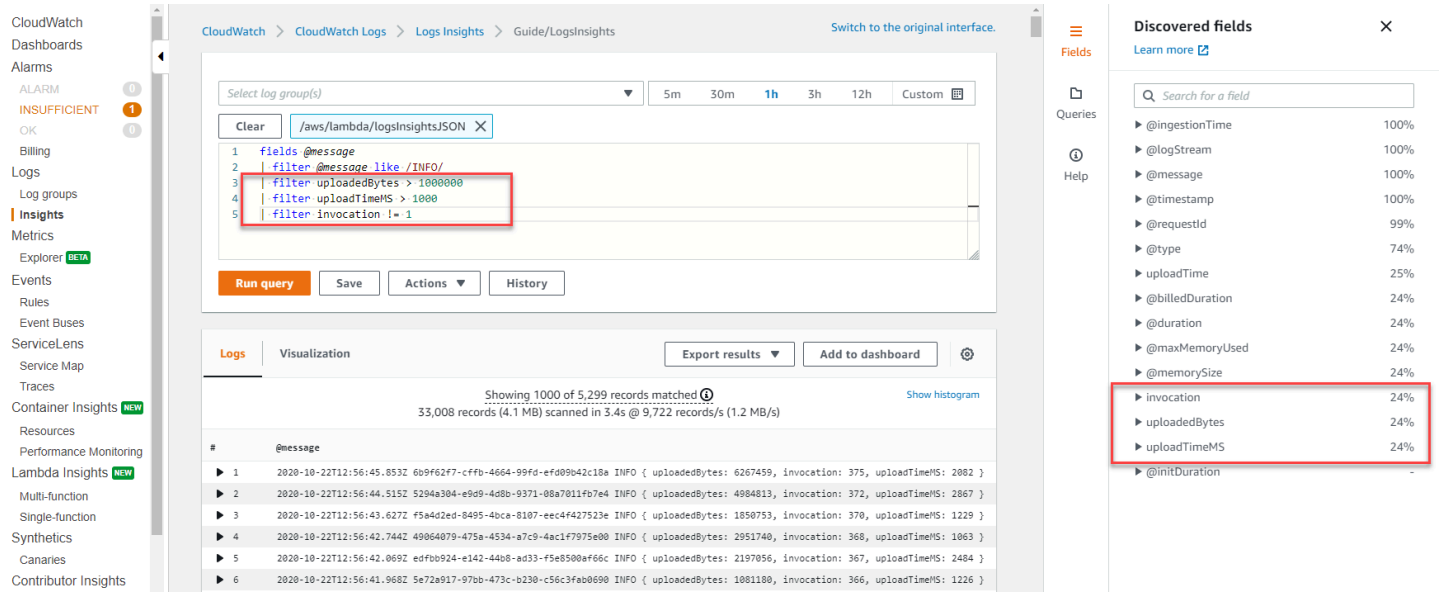
```
START RequestId: 22fda338-7b46-4c49-9531-efd6e8568480 Version: $LATEST
2020-10-22T11:35:59.216Z        22fda338-7b46-4c49-9531-efd6e8568480    INFO    { uploadedBytes: 5453396,
invocation: 5, uploadTimeMS: 4519 }
END RequestId: 22fda338-7b46-4c49-9531-efd6e8568480
REPORT RequestId: 22fda338-7b46-4c49-9531-efd6e8568480  Duration: 1.25 ms       Billed Duration: 100 ms Memory
Size: 128 MB    Max Memory Used: 65 MB
```

Die CloudWatch Logs Insights-Funktion erkennt automatisch Werte in der JSON-Ausgabe und analysiert die Nachrichten als Felder, ohne dass ein benutzerdefinierter Glob oder ein regulärer Ausdruck erforderlich ist. Mithilfe der JSON-structured Protokolle findet die folgende Abfrage Aufrufe, bei denen die hochgeladene Datei größer als 1 MB war, die Uploadzeit mehr als 1 Sekunde betrug und der Aufruf kein Kaltstart war:

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
```

```
| filter invocation != 1
```

Diese Abfrage könnte folgendes Ergebnis liefern:



Die in JSON erkannten Felder werden automatisch im Menü Erkannte Felder auf der rechten Seite ausgefüllt. Den vom Lambda-Dienst ausgegebenen Standardfeldern ist ein „@“ vorangestellt und Sie können diese Felder auf die gleiche Weise abfragen. Lambda-Logs enthalten immer die Felder @timestamp, @logStream, @message, @requestId, @duration, @billedDuration, @type, @maxMemoryUsed, @memorySize. Wenn X-Ray es für eine Ereignisfunktion aktiviert ist, enthalten die Protokolle auch @xray TraceId und @xraySegmentId.

Wenn eine AWS Ereignisquelle wie Amazon S3, Amazon SQS oder Amazon Ihre Event-Funktion EventBridge aufruft, wird das gesamte Ereignis der Event-Funktion als JSON-Objekteingabe zur Verfügung gestellt. Indem Sie dieses Ereignis in der ersten Zeile der Event-Funktion protokollieren, können Sie anschließend mithilfe CloudWatch von Logs Insights jedes der verschachtelten Felder abfragen.

Nützliche Insights-Abfragen

Die folgende Tabelle zeigt Beispiele für Insights-Abfragen, die für die Überwachung von Lambda-Ereignisfunktionen nützlich sein können.

Description	Beispiel für Abfragesyntax
Die letzten 100 Fehler	`fields Timestamp, LogLevel, Message`

Description	Beispiel für Abfragesyntax

	`| filter LogLevel == "ERR"`
	`| sort @timestamp desc`
	`| limit 100`

Die 100 Aufrufe mit den höchsten Abrechnungen	`filter @type = "REPORT"`
	`| fields @requestId, @billedDuration`
	`| sort by @billedDuration desc`
	`| limit 100`

Prozentualer Anteil der Kaltstarts an der Gesamtzahl der Aufrufe	`filter @type = "REPORT"`
	`| stats sum(strcontains(@message,`
	`"Init Duration"))/count(*) * 100 as`
	`coldStartPct, avg(@duration)`
	`by bin(5m)`

Perzentilbericht zur Lambda-Dauer	`filter @type = "REPORT"`
	`| stats`
	`avg(@billedDuration) as Average,`
	`percentile(@billedDuration, 99)`
	`as NinetyNinth,`
	`percentile(@billedDuration, 95)`
	`as NinetyFifth,`
	`percentile(@billedDuration, 90)`
	`as Ninetieth`
	`by bin(30m)`

Perzentilbericht zur Lambda-Speichernutzung	`filter @type="REPORT"`
	`| stats avg(@maxMemoryUsed/1024/102`
	`4) as mean_MemoryUsed,`
	`min(@maxMemoryUsed/1024/1024) as`
	`min_MemoryUsed,`
	`max(@maxMemoryUsed/1024/1024) as`
	`max_MemoryUsed,`
	`percentile(@maxMemoryUsed/1`
	`024/1024, 95) as Percentile95`

Description	Beispiel für Abfragesyntax

Aufrufe, die 100 % des zugewiesenen Speichers beanspruchen	`filter @type = "REPORT" and @maxMemoryUsed=@memorySize`
	`| stats`
	`count_distinct(@requestId)`
	`by bin(30m)`

Durchschnittlicher Speicherverbrauch bei Aufrufen	`avgMemoryUsedPERC,`
	`avg(@billedDuration) as avgDurationMS`
	`by bin(5m)`

Visualisierung von Speicherstatistiken	`filter @type = "REPORT"`
	`| stats`
	`max(@maxMemoryUsed / 1024 / 1024) as maxMemMB,`
	`avg(@maxMemoryUsed / 1024 / 1024) as avgMemMB,`
	`min(@maxMemoryUsed / 1024 / 1024) as minMemMB,`
	`(avg(@maxMemoryUsed / 1024 / 1024) / max(@memorySize / 1024 / 1024)) * 100 as avgMemUsedPct,`
	`avg(@billedDuration) as avgDurationMS`
	`by bin(30m)`

Aufrufe, bei denen Lambda beendet wurde	`filter @message like /Process exited/` `| stats count() by bin(30m)`
Aufrufe mit Zeitüberschreitung	`filter @message like /Task timed out/` `| stats count() by bin(30m)`

Description	Beispiel für Abfragesyntax
Latenzbericht	`filter @type = "REPORT" | stats avg(@duration), max(@duration), min(@duration) by bin(5m)`
Over-provisioned Speicher	`filter @type = "REPORT" | stats max(@memorySize / 1024 / 1024) as provisonedMemMB, min(@maxMemoryUsed / 1024 / 1024) as smallestMemReqMB, avg(@maxMemoryUsed / 1024 / 1024) as avgMemUsedMB, max(@maxMemoryUsed / 1024 / 1024) as maxMemUsedMB, provisonedMemMB - maxMemUsedMB as overProvisionedMB`

Protokollvisualisierung und Dashboards

Für jede CloudWatch Logs Insights-Abfrage können Sie die Ergebnisse im Markdown- oder CSV-Format exportieren. In einigen Fällen kann es sinnvoller sein, Visualisierungen aus Abfragen zu erstellen, sofern mindestens eine Aggregationsfunktion vorhanden ist. Mit der `stats` Funktion können Sie Aggregationen und Gruppierungen definieren.

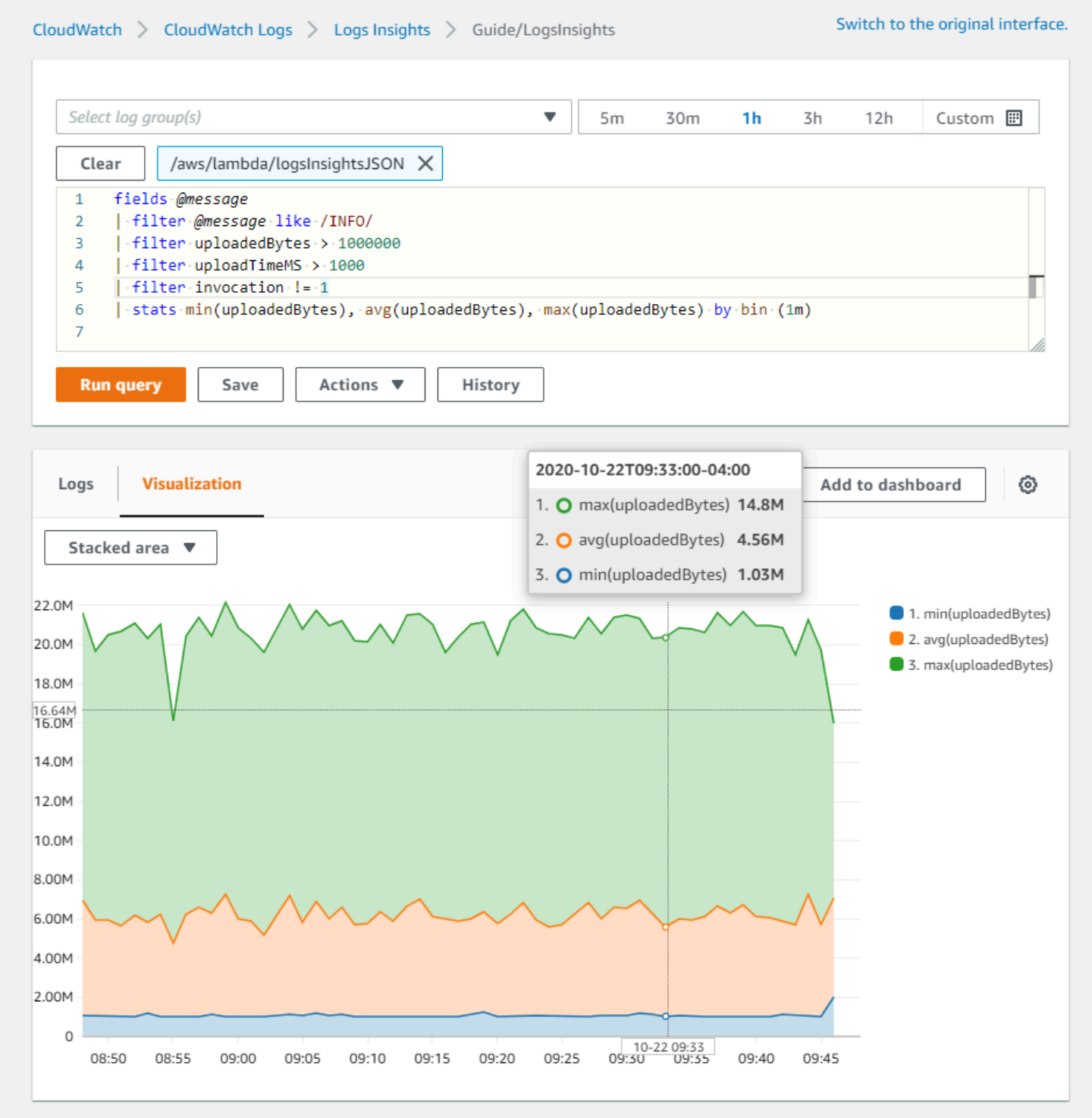
Das vorherige logInsightsJSON-Beispiel hat nach Upload-Größe und Upload-Zeit gefiltert und erste Aufrufe ausgeschlossen. Dies führte zu einer Datentabelle. Für die Überwachung eines Produktionssystems kann es sinnvoller sein, minimale, maximale und durchschnittliche Dateigrößen zu visualisieren, um Ausreißer zu finden. Wenden Sie dazu die Statistikfunktion mit den erforderlichen Aggregaten an und gruppieren Sie nach einem Zeitwert, z. B. jede Minute:

Nehmen Sie beispielsweise die folgende Abfrage. Dies ist dieselbe Beispielabfrage wie im Abschnitt the section called “JSON-strukturierte Protokollierung”, jedoch mit zusätzlichen Aggregationsfunktionen:

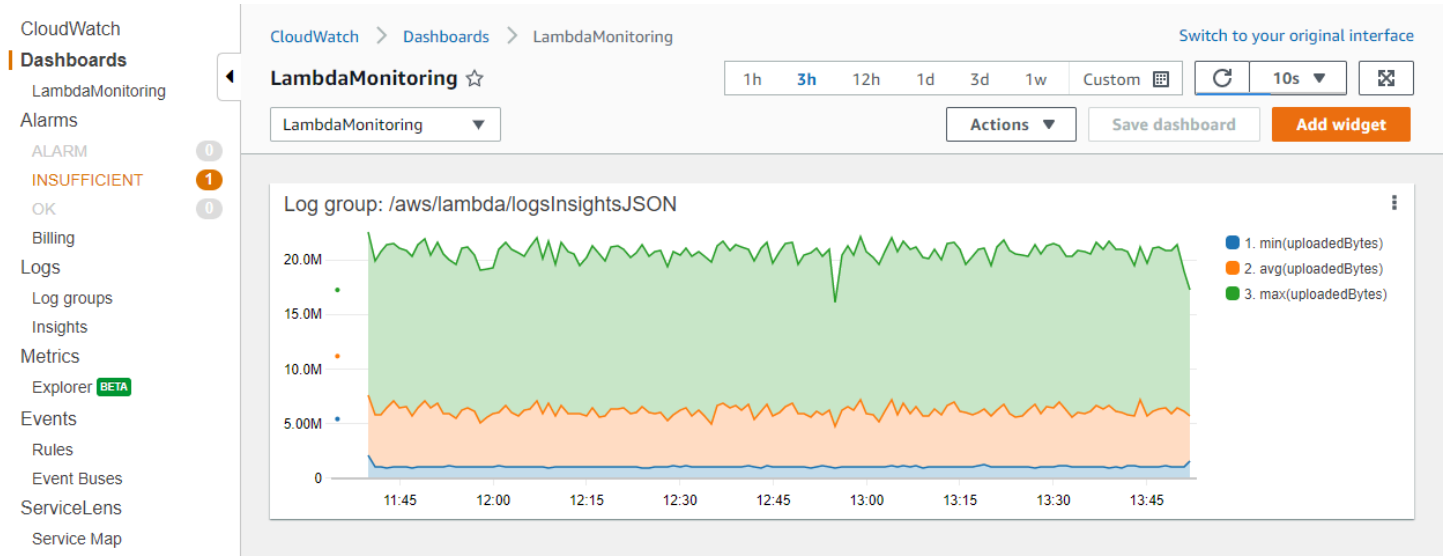
```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
```

```
| filter uploadTimeMS > 1000
| filter invocation != 1
| stats min(uploadedBytes), avg(uploadedBytes), max(uploadedBytes) by bin (1m)
```

Wir haben diese Aggregate aufgenommen, weil es sinnvoller sein könnte, minimale, maximale und durchschnittliche Dateigrößen zu visualisieren, um Ausreißer zu finden. Sie können die Ergebnisse auf der Registerkarte Visualisierung einsehen:



Nachdem Sie die Erstellung der Visualisierung abgeschlossen haben, können Sie das Diagramm optional zu einem CloudWatch Dashboard hinzufügen. Wählen Sie dazu oberhalb der Visualisierung die Option Zum Dashboard hinzufügen aus. Dadurch wird die Abfrage als Widget hinzugefügt und Sie können automatische Aktualisierungsintervalle auswählen, was die kontinuierliche Überwachung der Ergebnisse erleichtert:



Lambda Event Function-Protokolle an Firehose senden

Die Lambda-Konsole bietet jetzt die Möglichkeit, Event Function Logs an Firehose zu senden. Dies ermöglicht das Echtzeit-Streaming Ihrer Protokolle an verschiedene von Firehose unterstützte Ziele, einschließlich Analysetools von Drittanbietern und benutzerdefinierten Endpunkten.

> **Note**
>
> Sie können Lambda Event Function-Protokolle so konfigurieren, dass sie über die Lambda-Konsole, AWS CLI, AWS CloudFormation und alle SDKs an Firehose gesendet werden. AWS

Preisgestaltung

Einzelheiten zu den Preisen finden Sie unter CloudWatch Amazon-Preise.

Erforderliche Berechtigungen für das Firehose-Protokollziel

Wenn Sie die Lambda-Konsole verwenden, um Firehose als Protokollziel Ihrer Event-Funktion zu konfigurieren, benötigen Sie:

1. Die erforderlichen IAM-Berechtigungen für die Verwendung von CloudWatch Logs mit Lambda.
2. Sie müssen Abonnementfilter mit Firehose einrichten. Dieser Filter definiert, welche Protokollereignisse für Ihren Firehose-Stream bereitgestellt werden.

Lambda Event Function-Protokolle an Firehose senden

In der Lambda-Konsole können Sie Event Function Logs direkt an Firehose senden, nachdem Sie eine neue Event Function erstellt haben. Führen Sie dazu diese Schritte aus:

1. Melden Sie sich bei der AWS Management Console an und öffnen Sie die Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie die Registerkarte Überwachungs- und Betriebs-Tools aus.
5. Wählen Sie im Abschnitt „Protokollierungskonfiguration“ die Option Bearbeiten aus.
6. Wählen Sie im Abschnitt „Protokollinhalt“ ein Protokollformat aus.
7. Führen Sie im Abschnitt „Protokollziel“ die folgenden Schritte aus:
   a. Wählen Sie einen Zielservice aus.
   b. Wählen Sie Neue Protokollgruppe erstellen oder Vorhandene Protokollgruppe aus.

      > **Note**
      >
      > Wenn Sie eine vorhandene Protokollgruppe für ein Firehose-Ziel auswählen, stellen Sie sicher, dass es sich bei der ausgewählten Protokollgruppe um einen `Delivery`-Protokollgruppentyp handelt.

   c. Wählen Sie einen Firehose-Stream aus.
   d. Die CloudWatch `Delivery` Protokollgruppe wird angezeigt.
8. Wählen Sie Speichern.

> **Note**
>
> Wenn die in der Konsole angegebene IAM-Rolle nicht über die erforderlichen Berechtigungen verfügt, schlägt das Ziel-Setup fehl. Um dieses Problem zu beheben, finden Sie unter „Erforderliche Berechtigungen für das Firehose-Protokollziel“ die erforderlichen Berechtigungen.

Cross-Account Protokollierung

Sie können Lambda so konfigurieren, dass Protokolle in einem anderen AWS Konto an den Firehose-Lieferstream gesendet werden. Dies erfordert die Einrichtung eines Ziels und die Konfiguration der entsprechenden Berechtigungen in beiden Konten.

Detaillierte Anweisungen zur Einrichtung der kontoübergreifenden Protokollierung, einschließlich der erforderlichen IAM-Rollen und -Richtlinien, finden Sie unter Ein neues kontoübergreifendes Abonnement einrichten in der Logs-Dokumentation. CloudWatch

Lambda Event Function-Protokolle an Amazon S3 senden

Sie können Ihre Lambda-Ereignisfunktion so konfigurieren, dass Protokolle mithilfe der Lambda-Konsole direkt an Amazon S3 gesendet werden. Dieses Feature bietet eine kostengünstige Lösung für die langfristige Speicherung von Protokollen und ermöglicht leistungsstarke Analyseoptionen mithilfe von Services wie Athena.

> **Note**
>
> Sie können Lambda Event Function-Protokolle so konfigurieren, dass sie mit der Lambda-Konsole, AWS CLI AWS CloudFormation, und allen AWS SDKs an Amazon S3 gesendet werden.

Preisgestaltung

Einzelheiten zu den Preisen finden Sie unter CloudWatch Amazon-Preise.

Erforderliche Berechtigungen für das Amazon-S3-Protokollziel

Wenn Sie die Lambda-Konsole verwenden, um Amazon S3 als Protokollziel Ihrer Event Function zu konfigurieren, benötigen Sie:

1. Die erforderlichen IAM-Berechtigungen für die Verwendung von CloudWatch Logs mit Lambda.
2. Zu. Richten Sie einen CloudWatch Logs-Abonnementfilter ein, um Lambda Event Function-Logs an Amazon S3 zu senden Dieser Filter definiert, welche Protokollereignisse für Ihren Amazon-S3-Bucket bereitgestellt werden.

Richten Sie einen CloudWatch Logs-Abonnementfilter ein, um Lambda Event Function-Logs an Amazon S3 zu senden

Um Logs von CloudWatch Logs an Amazon S3 zu senden, müssen Sie einen Abonnementfilter erstellen. Dieser Filter definiert, welche Protokollereignisse für Ihren Amazon-S3-Bucket bereitgestellt werden. Ihr Amazon-S3-Bucket muss sich in derselben Region befinden wie Ihre Protokollgruppe.

So erstellen Sie einen Abonnementfilter für Amazon S3

1. Erstellen Sie einen Amazon-Simple-Storage-Service-(Amazon S3)-Bucket. Wir empfehlen Ihnen, einen Bucket zu verwenden, der speziell für CloudWatch Logs erstellt wurde. Wenn Sie jedoch einen vorhandenen Bucket verwenden möchten, gehen Sie direkt zu Schritt 2.

   Führen Sie den folgenden Befehl aus, und ersetzen Sie die Platzhalter-Region mit der Region, die Sie verwenden möchten:

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-bucket2 --create-bucket-configuration
    LocationConstraint=region
   ```

   > **Note**
   >
   > `amzn-s3-demo-bucket2` ist ein Beispiel für den Namen eines Amazon-S3-Buckets. Er ist reserviert. Damit dieses Verfahren funktioniert, müssen Sie ihn durch den eindeutigen Namen Ihres Amazon-S3-Buckets ersetzen.

   Das Folgende ist eine Beispielausgabe:

   ```
   {
       "Location": "/amzn-s3-demo-bucket2"
   }
   ```

2. Erstellen Sie die IAM-Rolle, die CloudWatch Logs die Berechtigung erteilt, Daten in Ihren Amazon S3 S3-Bucket zu legen. Diese Richtlinie beinhaltet den Kontextschlüssel aws: SourceArn global condition, um das Sicherheitsproblem „Confused Deputy“ zu verhindern. Weitere Informationen finden Sie unter Confused-Deputy-Prävention.

   a. Verwenden Sie einen Text-Editor zum Erstellen einer Vertrauensrichtlinie in einer Datei `~/TrustPolicyForCWL.json` wie folgt:

```
{
    "Statement": {
        "Effect": "Allow",
        "Principal": { "Service": "logs.amazonaws.com" },
        "Condition": {
            "StringLike": {
                "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
            }
         },
        "Action": "sts:AssumeRole"
    }
}
```

b. Verwenden Sie den Befehl create-role, um die IAM-Rolle zu erstellen und die Datei mit der Vertrauensrichtlinie anzugeben. Notieren Sie sich den zurückgegebenen Role.Arn Wert, da Sie ihn in einem späteren Schritt benötigen:

```
aws iam create-role \
 --role-name CWLtoS3Role \
 --assume-role-policy-document file://~/TrustPolicyForCWL.json
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Statement": {
                "Action": "sts:AssumeRole",
                "Effect": "Allow",
                "Principal": {
                    "Service": "logs.amazonaws.com"
                },
                "Condition": {
                    "StringLike": {
                        "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
                    }
                }
            }
        },
        "RoleId": "AAOIIAH450GAB4HC5F431",
        "CreateDate": "2015-05-29T13:46:29.431Z",
        "RoleName": "CWLtoS3Role",
        "Path": "/",
        "Arn": "arn:aws:iam::123456789012:role/CWLtoS3Role"
    }
```

```
}
```

3. Erstellen Sie eine Berechtigungsrichtlinie, um zu definieren, welche Aktionen CloudWatch Logs auf Ihrem Konto ausführen können. Verwenden Sie zunächst einen Text-Editor zum Erstellen einer Berechtigungsrichtlinie in einer Datei `~/PermissionsForCWL.json`:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:PutObject"],
      "Resource": ["arn:aws:s3:::amzn-s3-demo-bucket2/*"]
    }
  ]
}
```

   Verknüpfen Sie die Berechtigungsrichtlinie mit dem folgenden Befehl `put-role-policy` mit der Rolle:

```
aws iam put-role-policy --role-name CWLtoS3Role --policy-name Permissions-Policy-For-
S3 --policy-document file://~/PermissionsForCWL.json
```

4. Erstellen Sie eine `Delivery`-Protokollgruppe oder verwenden Sie eine vorhandene `Delivery`-Protokollgruppe.

```
aws logs create-log-group --log-group-name my-logs --log-group-class DELIVERY --
region REGION_NAME
```

5. `PutSubscriptionFilter` zum Einrichten des Ziels

```
aws logs put-subscription-filter
--log-group-name my-logs
--filter-name my-lambda-delivery
--filter-pattern ""
--destination-arn arn:aws:s3:::amzn-s3-demo-bucket2
--role-arn arn:aws:iam::123456789012:role/CWLtoS3Role
--region REGION_NAME
```

Lambda Event Function-Protokolle an Amazon S3 senden

In der Lambda-Konsole können Sie Event Function Logs direkt an Amazon S3 senden, nachdem Sie eine neue Event Function erstellt haben. Führen Sie dazu diese Schritte aus:

1. Melden Sie sich bei der AWS Management Console an und öffnen Sie die Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Event-Funktion.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie die Registerkarte Überwachungs- und Betriebs-Tools aus.
5. Wählen Sie im Abschnitt „Protokollierungskonfiguration“ die Option Bearbeiten aus.
6. Wählen Sie im Abschnitt „Protokollinhalt“ ein Protokollformat aus.
7. Führen Sie im Abschnitt „Protokollziel“ die folgenden Schritte aus:
   a. Wählen Sie einen Zielservice aus.
   b. Wählen Sie Neue Protokollgruppe erstellen oder Vorhandene Protokollgruppe aus.

      > **Note**
      >
      > Wenn Sie eine vorhandene Protokollgruppe für ein Amazon-S3-Ziel auswählen, stellen Sie sicher, dass es sich bei der ausgewählten Protokollgruppe um einen `Delivery`-Protokollgruppentyp handelt.

   c. Wählen Sie einen Amazon S3 S3-Bucket als Ziel für Ihre Event Function-Logs aus.
   d. Die CloudWatch `Delivery` Protokollgruppe wird angezeigt.
8. Wählen Sie Speichern.

> **Note**
>
> Wenn die in der Konsole bereitgestellte IAM-Rolle nicht über die erforderlichen Berechtigungen verfügt, schlägt das Ziel-Setup fehl. Informationen zur Behebung dieses Problems finden Sie unter Erforderliche Berechtigungen für das Amazon-S3-Protokollziel.

Cross-Account Protokollierung

Sie können Lambda so konfigurieren, dass Protokolle an einen Amazon S3 S3-Bucket in einem anderen AWS Konto gesendet werden. Dies erfordert die Einrichtung eines Ziels und die Konfiguration der entsprechenden Berechtigungen in beiden Konten.

Detaillierte Anweisungen zur Einrichtung der kontoübergreifenden Protokollierung, einschließlich der erforderlichen IAM-Rollen und -Richtlinien, finden Sie unter Ein neues kontoübergreifendes Abonnement einrichten in der Logs-Dokumentation. CloudWatch

Protokollierung AWS Lambda API-Aufrufe mit AWS CloudTrail

AWS Lambda ist in einen Dienst integriert AWS CloudTrail, der eine Aufzeichnung der von einem Benutzer, einer Rolle oder einem ausgeführten Aktionen bereitstellt AWS-Service. CloudTrail erfasst API-Aufrufe für Lambda als Ereignisse. Zu den erfassten Aufrufen gehören Aufrufe über die Lambda-Konsole und Codeaufrufe der Lambda-API-Operationen. Anhand der von gesammelten Informationen können Sie die Anfrage CloudTrail, die an Lambda gestellt wurde, die IP-Adresse, von der aus die Anfrage gestellt wurde, wann sie gestellt wurde, und weitere Details ermitteln.

Jeder Ereignis- oder Protokolleintrag enthält Informationen zu dem Benutzer, der die Anforderung generiert hat. Die Identitätsinformationen unterstützen Sie bei der Ermittlung der folgenden Punkte:

- Ob die Anforderung mit Anmeldeinformationen des Root-Benutzers oder des Benutzers gestellt wurde.
- Die Anforderung wurde im Namen eines IAM-Identity-Center-Benutzers erstellt.
- Gibt an, ob die Anforderung mit temporären Sicherheitsanmeldeinformationen für eine Rolle oder einen Verbundbenutzer gesendet wurde.
- Ob die Anforderung aus einem anderen AWS-Service gesendet wurde.

CloudTrail ist in Ihrem aktiv AWS-Konto , wenn Sie das Konto erstellen, und Sie haben automatisch Zugriff auf den CloudTrail Eventverlauf. Der CloudTrail Ereignisverlauf bietet eine einsehbare, durchsuchbare, herunterladbare und unveränderliche Aufzeichnung der aufgezeichneten Verwaltungsereignisse der letzten 90 Tage in einem. AWS-Region Weitere Informationen finden Sie im AWS CloudTrail Benutzerhandbuch unter Arbeiten mit dem CloudTrail Ereignisverlauf. Für die Anzeige des Eventverlaufs CloudTrail fallen keine Gebühren an.

Für eine fortlaufende Aufzeichnung der Ereignisse in AWS-Konto den letzten 90 Tagen erstellen Sie einen Trail- oder CloudTrailLake-Event-Datenspeicher.

CloudTrail Pfade

> Ein Trail ermöglicht CloudTrail die Übermittlung von Protokolldateien an einen Amazon S3 S3-Bucket. Alle mit dem erstellten Pfade AWS-Managementkonsole sind regionsübergreifend. Sie können mithilfe von AWS CLI einen Einzel-Region- oder einen Multi-Region-Trail erstellen. Es wird empfohlen, einen Trail mit mehreren Regionen zu erstellen, da Sie alle Aktivitäten AWS-Regionen in Ihrem Konto erfassen. Wenn Sie einen Einzel-Region-Trail erstellen, können Sie nur die Ereignisse anzeigen, die im AWS-Region des Trails protokolliert wurden. Weitere

Informationen zu Trails finden Sie unter Erstellen eines Trails für Ihr AWS-Konto und Erstellen eines Trails für eine Organisation im AWS CloudTrail -Benutzerhandbuch.

Sie können eine Kopie Ihrer laufenden Verwaltungsereignisse kostenlos an Ihren Amazon S3 S3-Bucket senden, CloudTrail indem Sie einen Trail erstellen. Es fallen jedoch Amazon S3 S3-Speichergebühren an. Weitere Informationen zur CloudTrail Preisgestaltung finden Sie unter AWS CloudTrail Preise. Informationen zu Amazon-S3-Preisen finden Sie unter Amazon S3 – Preise.

CloudTrail Datenspeicher für Ereignisse in Lake

CloudTrail Mit Lake können Sie SQL-based Abfragen zu Ihren Ereignissen ausführen. CloudTrail Lake konvertiert bestehende Ereignisse im zeilenbasierten JSON-Format in das Apache ORC-Format. ORC ist ein spaltenförmiges Speicherformat, das für den schnellen Abruf von Daten optimiert ist. Die Ereignisse werden in Ereignisdatenspeichern zusammengefasst, bei denen es sich um unveränderliche Sammlungen von Ereignissen handelt, die auf Kriterien basieren, die Sie mit Hilfe von erweiterten Ereignisselektoren auswählen. Die Selektoren, die Sie auf einen Ereignisdatenspeicher anwenden, steuern, welche Ereignisse bestehen bleiben und für Sie zur Abfrage verfügbar sind. Weitere Informationen zu CloudTrail Lake finden Sie unter Arbeiten mit AWS CloudTrail Lake im AWS CloudTrail Benutzerhandbuch.

CloudTrail Für das Speichern und Abfragen von Ereignisdaten in Lake fallen Kosten an. Beim Erstellen eines Ereignisdatenspeichers wählen Sie die Preisoption aus, die für den Ereignisdatenspeicher genutzt werden soll. Die Preisoption bestimmt die Kosten für die Erfassung und Speicherung von Ereignissen sowie die standardmäßige und maximale Aufbewahrungsdauer für den Ereignisdatenspeicher. Weitere Informationen zur CloudTrail Preisgestaltung finden Sie unter AWS CloudTrail Preise.

Lambda-Datenereignisse in CloudTrail

Datenereignisse liefern Informationen über die Ressourcenoperationen, die auf oder in einer Ressource ausgeführt werden (z. B. Lesen oder Schreiben in ein Amazon-S3-Objekt). Sie werden auch als Vorgänge auf Datenebene bezeichnet. Datenereignisse sind oft Aktivitäten mit hohem Volume. Standardmäßig werden die meisten Datenereignisse CloudTrail nicht protokolliert, und der CloudTrail Ereignisverlauf zeichnet sie nicht auf.

Ein CloudTrail Datenereignis, das standardmäßig für unterstützte Dienste protokolliert wird, ist`LambdaESMDisabled`. Weitere Informationen zur Verwendung dieses Ereignisses zur Behebung von Problemen mit Lambda-Zuordnungen von Ereignisquellen finden Sie unter the section called “Wird CloudTrail zur Fehlerbehebung bei deaktivierten Lambda-Ereignisquellen verwendet”.

Für Datenereignisse werden zusätzliche Gebühren fällig. Weitere Informationen zur CloudTrail Preisgestaltung finden Sie unter AWS CloudTrail Preisgestaltung.

Sie können Datenereignisse für den `AWS::Lambda::Function` Ressourcentyp mithilfe der CloudTrail Konsole oder CloudTrail API-Operationen protokollieren. AWS CLI Weitere Informationen zum Protokollieren von Datenereignissen finden Sie unter Protokollieren von Datenereignissen mit dem AWS-Managementkonsole und Protokollieren von Datenereignissen mit dem AWS Command Line Interface im AWS CloudTrail -Benutzerhandbuch.

In der folgenden Tabelle ist der Lambda-Ressourcentyp aufgeführt, für die Sie Datenereignisse protokollieren können. In der Spalte Datenereignistyp (Konsole) wird der Wert angezeigt, den Sie in der Liste Datenereignistyp auf der CloudTrail Konsole auswählen können. In der Wertspalte resources.type wird der `resources.type` Wert angezeigt, den Sie angeben würden, wenn Sie erweiterte Event-Selektoren mithilfe der AWS CLI APIs oder konfigurieren würden. CloudTrail In der CloudTrail Spalte „Protokollierte Daten-APIs“ werden die API-Aufrufe angezeigt, die CloudTrail für den Ressourcentyp protokolliert wurden.

Typ des Datenereignisses (Konsole)	resources.type-Wert	Daten-APIs, bei denen angemeldet wurde CloudTrail
Lambda	`AWS::Lambda::Function`	Invoke

Sie können erweiterte Event-Selektoren so konfigurieren, dass sie nach den Feldern `eventName`, `readOnly` und `resources.`ARN filtern, sodass nur die Ereignisse protokolliert werden, die für Sie wichtig sind. Das folgende Beispiel zeigt die JSON-Ansicht einer Datenereigniskonfiguration, die nur Ereignisse für eine bestimmte Ereignisfunktion protokolliert. Weitere Informationen zu diesen Kontingenten finden Sie unter AdvancedFieldSelector in derAWS CloudTrail -API-Referenz.

```
[
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
```

```
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
        ]
      }
    ]
  }
]
```

Lambda-Management-Ereignisse in CloudTrail

Verwaltungsereignisse enthalten Informationen über Verwaltungsvorgänge, die an Ressourcen in Ihrem AWS-Konto ausgeführt werden. Sie werden auch als Vorgänge auf Steuerebene bezeichnet. CloudTrail Protokolliert standardmäßig Verwaltungsereignisse.

Lambda unterstützt die Protokollierung der folgenden Aktionen als Verwaltungsereignisse in CloudTrail Protokolldateien.

> **Note**
>
> In der CloudTrail Protokolldatei `eventName` können sie Datums- und Versionsinformationen enthalten, sie beziehen sich jedoch immer noch auf dieselbe öffentliche API-Aktion. Zum Beispiel erscheint die Aktion `GetFunction` als `GetFunction20150331v2`. Der folgende Liste gibt an, wann sich der Ereignisname vom API-Aktionsnamen unterscheidet.

- AddLayerVersionPermission
- AddPermission(Name des Ereignisses:`AddPermission20150331v2`)
- CreateAlias(Name des Ereignisses:`CreateAlias20150331`)
- CreateEventSourceMapping(Name des Ereignisses:`CreateEventSourceMapping20150331`)
- CreateFunction(Name des Ereignisses:`CreateFunction20150331`)

(Die `Environment`- und `ZipFile`-Parameter fließen nicht in die CloudTrail -Protokolle für `CreateFunction` mit ein.)

- CreateFunctionUrlConfig
- DeleteAlias(Name des Ereignisses:`DeleteAlias20150331`)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping(Name des Ereignisses:`DeleteEventSourceMapping20150331`)
- DeleteFunction(Name des Ereignisses:`DeleteFunction20150331`)
- DeleteFunctionConcurrency(Name des Ereignisses:`DeleteFunctionConcurrency20171031`)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias(Name des Ereignisses:`GetAlias20150331`)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion(Name des Ereignisses:`PublishLayerVersion20181031`)

  (Der `ZipFile` Parameter wird in den CloudTrail Protokollen für weggelassen`PublishLayerVersion`.)
- PublishVersion(Name des Ereignisses:`PublishVersion20150331`)
- PutFunctionConcurrency(Name des Ereignisses:`PutFunctionConcurrency20171031`)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- PutRuntimeManagementConfig
- RemovePermission(Name des Ereignisses:`RemovePermission20150331v2`)

- TagResource(Name des Ereignisses:`TagResource20170331v2`)
- UntagResource(Name des Ereignisses:`UntagResource20170331v2`)
- UpdateAlias(Name des Ereignisses:`UpdateAlias20150331`)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping(Name des Ereignisses:`UpdateEventSourceMapping20150331`)
- UpdateFunctionCode(Name des Ereignisses:`UpdateFunctionCode20150331v2`)

  (Der `ZipFile` Parameter wird in den CloudTrail Protokollen für weggelassen`UpdateFunctionCode`.)
- UpdateFunctionConfiguration(Name des Ereignisses:`UpdateFunctionConfiguration20150331v2`)

  (Der `Environment` Parameter wird in den CloudTrail Protokollen für weggelassen`UpdateFunctionConfiguration`.)
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

Wird CloudTrail zur Fehlerbehebung bei deaktivierten Lambda-Ereignisquellen verwendet

Wenn Sie den Status einer Ereignisquellenzuordnung mithilfe der UpdateEventSourceMappingAPI-Aktion ändern, wird der API-Aufruf als Verwaltungsereignis protokolliert. CloudTrail Zuordnungen von Ereignisquellen können aufgrund von Fehlern auch direkt in den `Disabled`-Status übergehen.

Für die folgenden Dienste veröffentlicht Lambda das `LambdaESMDisabled` Datenereignis, CloudTrail wenn Ihre Ereignisquelle in den Status Deaktiviert übergeht:

- Amazon-Simple-Queue-Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda unterstützt dieses Ereignis nicht für andere Ereignisquellen-Zuordnungstypen.

Um Benachrichtigungen zu erhalten, wenn Ereignisquellenzuordnungen für unterstützte Dienste in den `Disabled` Status wechseln, richten Sie in Amazon einen Alarm ein, der das

`LambdaESMDisabled` CloudTrail Ereignis CloudWatch verwendet. Weitere Informationen zum Einrichten eines CloudWatch Alarms finden Sie unter CloudWatch Alarme für CloudTrail Ereignisse erstellen: Beispiele.

Die `serviceEventDetails`-Entität in der `LambdaESMDisabled`-Ereignismeldung enthält einen der folgenden Fehlercodes.

`RESOURCE_NOT_FOUND`

Die in der Anforderung angegebene Ressource ist nicht vorhanden.

`FUNCTION_NOT_FOUND`

Die an die Ereignisquelle angehängte Ereignisfunktion ist nicht vorhanden.

`REGION_NAME_NOT_VALID`

Ein für die Ereignisquelle oder Ereignisfunktion bereitgestellter Regionsname ist ungültig.

`AUTHORIZATION_ERROR`

Es wurden keine Berechtigungen festgelegt oder sie wurden falsch konfiguriert.

`FUNCTION_IN_FAILED_STATE`

Der Code der Ereignisfunktion kann nicht kompiliert werden, es ist eine nicht behebbare Ausnahme aufgetreten, oder es ist eine fehlerhafte Bereitstellung aufgetreten.

Beispiele für Lambda-Ereignisse

Ein Ereignis stellt eine einzelne Anforderung aus einer beliebigen Quelle dar und enthält Informationen über den angeforderten API-Vorgang, Datum und Uhrzeit des Vorgangs, Anforderungsparameter usw. CloudTrail Protokolldateien sind kein geordneter Stack-Trace der öffentlichen API-Aufrufe, sodass Ereignisse nicht in einer bestimmten Reihenfolge angezeigt werden.

Das folgende Beispiel zeigt CloudTrail Protokolleinträge für die `DeleteFunction` Aktionen `GetFunction` und.

> **Note**
>
> Die `eventName` kann Datums- und Versionsinformationen enthalten, wie z. B. die `"GetFunction20150331"`, aber sie bezieht sich immer noch auf dieselbe öffentliche API.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
```

```
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
  ]
}
```

Informationen zu CloudTrail Datensatzinhalten finden Sie im AWS CloudTrail Benutzerhandbuch unter CloudTrailDatensatzinhalte.

Visualisieren Sie Aufrufe von Lambda-Ereignisfunktionen mit AWS X-Ray

Sie können AWS X-Ray es verwenden, um die Komponenten Ihrer Anwendung zu visualisieren, Leistungsengpässe zu identifizieren und Anfragen zu beheben, die zu einem Fehler geführt haben. Ihre Lambda-Ereignisfunktionen senden Trace-Daten an und X-Ray X-Ray verarbeiten die Daten, um eine Service-Map und durchsuchbare Trace-Zusammenfassungen zu generieren.

Lambda unterstützt zwei Verfolgungsmodi für X-Ray: `Active` und. `PassThrough` Mit `Active` der Ablaufverfolgung erstellt Lambda automatisch Trace-Segmente für Event-Function-Aufrufe und sendet sie an. X-Ray `PassThrough`mode hingegen gibt den Ablaufverfolgungskontext einfach an nachgeschaltete Dienste weiter.

Wenn Sie die `Active` Ablaufverfolgung für Ihre Event-Funktion aktiviert haben, sendet Lambda automatisch Traces an X-Ray für gesampelte Anfragen. In der Regel entscheidet ein Upstream-Service wie Amazon API Gateway oder eine auf Amazon EC2 gehostete Anwendung, die mit dem X-Ray SDK instrumentiert ist, ob eingehende Anfragen verfolgt werden sollen, und fügt dann diese Stichprobenentscheidung als Tracing-Header hinzu. Lambda verwendet diesen Header, um zu entscheiden, ob Nachverfolgungen gesendet werden sollen oder nicht. Traces von Upstream-Nachrichtenproduzenten, wie Amazon SQS, werden automatisch mit Traces von nachgeschalteten Lambda-Ereignisfunktionen verknüpft, wodurch eine durchgängige Ansicht der gesamten Anwendung entsteht. Weitere Informationen finden Sie unter Ablaufverfolgung ereignisgesteuerter Anwendungen im AWS X-Ray -Entwicklerhandbuch.

> **Note**
>
> X-Ray Die Ablaufverfolgung wird derzeit nicht für Lambda Event Functions mit Amazon Managed Streaming for Apache Kafka (Amazon MSK), selbstverwaltetem Apache Kafka, Amazon MQ mit ActiveMQ und RabbitMQ oder Amazon DocumentDB DocumentDB-Ereignisquellenzuordnungen unterstützt.

Gehen Sie wie folgt vor, um die aktive Ablaufverfolgung in Ihrer Lambda-Ereignisfunktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.

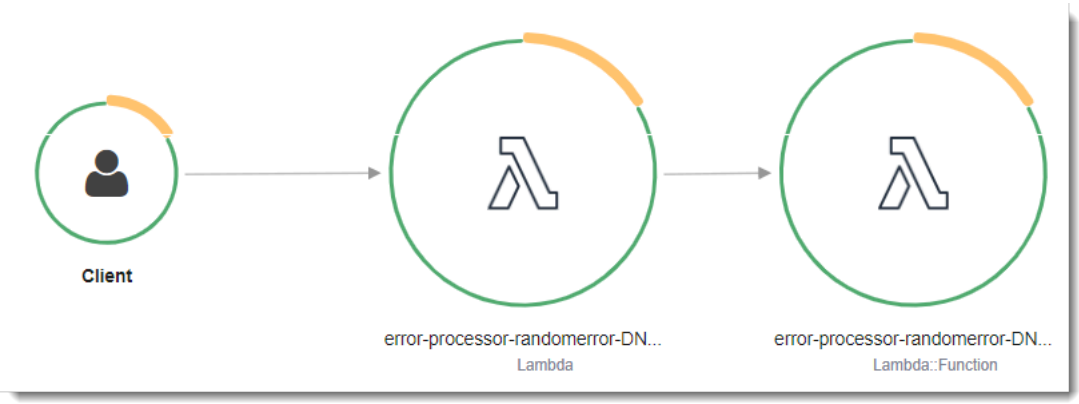
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Ihre Event-Funktion benötigt die Erlaubnis, Trace-Daten hochzuladen. X-Ray Wenn Sie die Ablaufverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Event-Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

X-Ray verfolgt nicht alle Anfragen zu Ihrer Anwendung. X-Ray wendet einen Stichprobenalgorithmus an, um sicherzustellen, dass die Rückverfolgung effizient ist und gleichzeitig eine repräsentative Stichprobe aller Anfragen bereitgestellt wird. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray Samplingrate für Ihre Funktionen nicht konfigurieren.

X-Ray Spuren verstehen

In X-Ray einer Ablaufverfolgung werden Informationen über eine Anfrage aufgezeichnet, die von einem oder mehreren Diensten bearbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten steht für Ihre spezifische Lambda-Ereignisfunktion.

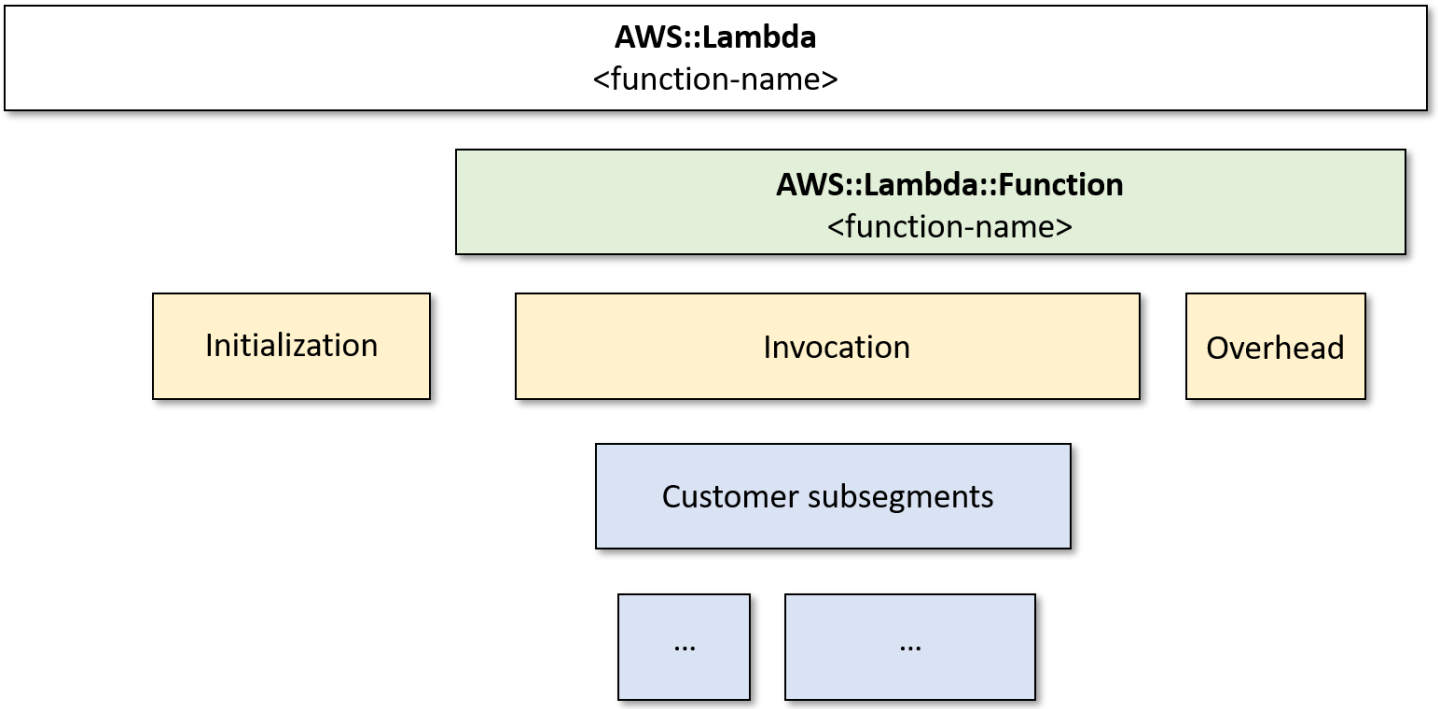
Das für den Lambda-Dienst `AWS::Lambda` aufgezeichnete Segment umfasst alle Schritte, die zur Vorbereitung der Lambda-Ausführungsumgebung erforderlich sind. Dazu gehören das Scheduling der MicroVM, das Erstellen oder Aufheben des Einfrierens einer Ausführungsumgebung mit den von Ihnen konfigurierten Ressourcen sowie das Herunterladen Ihres Funktionscodes und aller Schichten.

Das `AWS::Lambda::Function` Segment ist für die Arbeit vorgesehen, die von der Event-Funktion geleistet wird.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können Sie geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollnachrichten und Trace-Segmenten feststellen, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Diese Änderung betrifft die Teilsegmente des Funktionssegments. In den folgenden Abschnitten werden sowohl das alte als auch das neue Format für diese Teilsegmente beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Old-style AWS X-Ray Lambda-Segmentstruktur

Die alte X-Ray Struktur für das `AWS::Lambda` Segment sieht wie folgt aus:



In diesem Format hat das Funktionssegment Untersegmente für `Initialization`, `Invocation` und `Overhead`. Nur für SnapStart gibt es auch ein Teilsegment `Restore` (in dieser Abbildung nicht dargestellt).

Das `Initialization`-Teilsegment stellt die Init-Phase des Lebenszyklus der Lambda-Ausführungsumgebung dar. In dieser Phase initialisiert Lambda Erweiterungen, initialisiert die Laufzeit und führt den Initialisierungscode der Funktion aus.

Das `Invocation`-Teilsegment stellt die Aufrufphase dar, in welcher der Lambda-Funktionshandler aufgerufen wird. Dies beginnt mit der Laufzeit- und Erweiterungsregistrierung und endet, wenn die Laufzeit bereit ist, die Antwort zu senden.

(SnapStart Nur Lambda) Das `Restore` Untersegment zeigt die Zeit, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und alle Runtime-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird im `Restore`-Untersegment erfasst. Die Zeit, die Sie außerhalb der MicroVM für die Wiederherstellung eines Snapshots aufgewendet haben, wird Ihnen nicht in Rechnung gestellt.

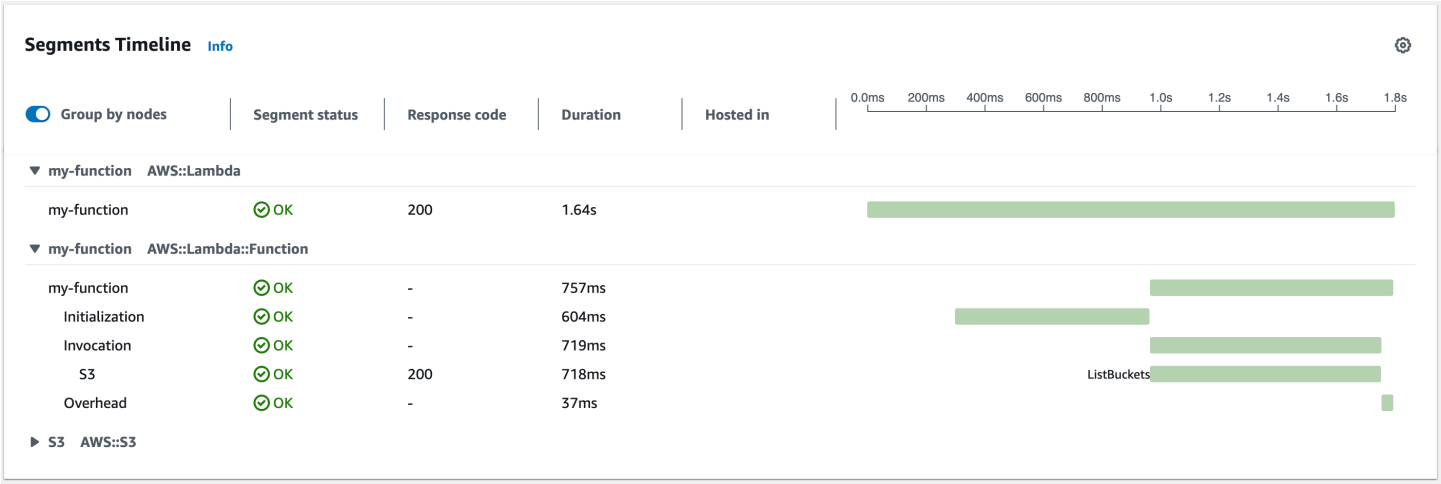
Das `Overhead`-Teilsegment stellt die Phase dar, die zwischen dem Zeitpunkt, zu dem die Laufzeit die Antwort sendet, und dem Signal für den nächsten Aufruf auftritt. Während dieser Zeit beendet die

Laufzeit alle Aufgaben im Zusammenhang mit einem Aufruf und bereitet sich auf das Einfrieren der Sandbox vor.

> **Important**
>
> Sie können das X-Ray SDK verwenden, um das `Invocation` Untersegment um zusätzliche Untersegmente für Downstream-Aufrufe, Anmerkungen und Metadaten zu erweitern. Sie können nicht direkt auf das Funktionssegment zugreifen oder Arbeiten außerhalb des Handler-Aufruffunktionsbereichs aufzeichnen.

Weitere Informationen zu den Phasen der Lambda-Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Ein Beispiel für eine Ablaufverfolgung, die die X-Ray Struktur im alten Stil verwendet, ist in der folgenden Abbildung dargestellt.



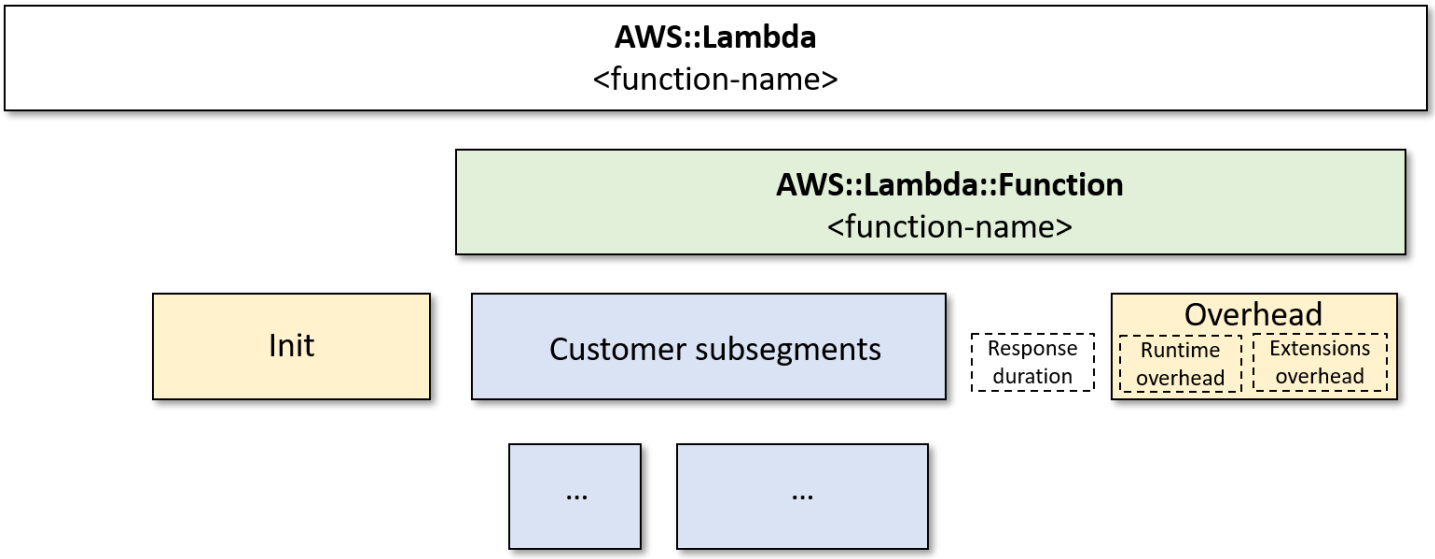
Beachten Sie die beiden Segmente im Beispiel. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function` Segment einen Fehler anzeigt, ist bei Ihrer Event-Funktion ein Problem aufgetreten.

> **Note**
>
> Gelegentlich stellen Sie in Ihren X-Ray Traces möglicherweise eine große Lücke zwischen der Initialisierung der Funktion und der Aufrufphase fest. Bei Funktionen, die bereitgestellte Gleichzeitigkeit verwenden, liegt dass daran, dass Lambda Ihre Funktions-Instances lange

vor dem Aufruf initialisiert. Für Eventfunktionen, die unreservierte (On-Demand-Parallelität) verwenden, initialisiert Lambda möglicherweise proaktiv eine Funktionsinstanz, auch wenn kein Aufruf erfolgt. Visuell werden diese beiden Fälle als Zeitlücke zwischen der Initialisierungs- und der Aufrufphase angezeigt.

New-style AWS X-Ray Lambda-Segmentstruktur

Die X-Ray Struktur im neuen Stil für das `AWS::Lambda` Segment sieht wie folgt aus:

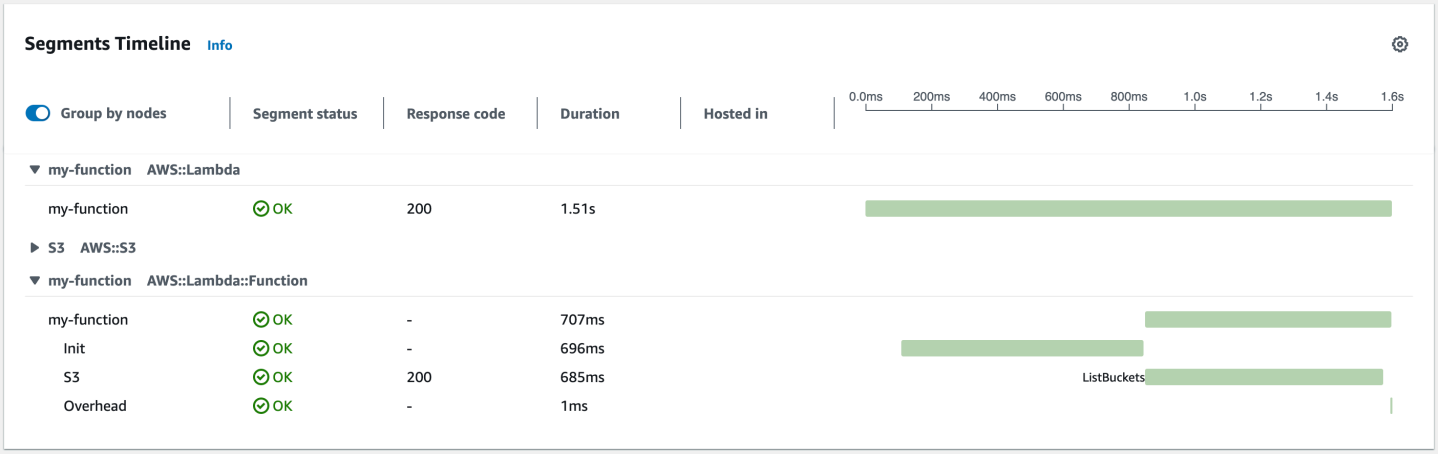


In diesem neuen Format stellt das Teilsegment `Init` wie bisher die Init-Phase des Lebenszyklus der Lambda-Ausführungsumgebung dar.

Im neuen Format gibt es kein Aufrufsegment. Stattdessen werden Kundenuntersegmente direkt an das `AWS::Lambda::Function`-Segment angehängt. Dieses Segment enthält die folgenden Metriken als Anmerkungen:

- `aws.responseLatency` – die Zeit, die benötigt wurde, bis die Funktion ausgeführt wurde
- `aws.responseDuration` – die Zeit, die benötigt wurde, um die Antwort an den Kunden zu übermitteln
- `aws.runtimeOverhead` – die zusätzliche Zeit, die die Laufzeit bis zum Abschluss benötigt hat
- `aws.extensionOverhead` – die zusätzliche Zeit, die die Erweiterungen zum Abschluss benötigt hat

Ein Beispiel für eine Ablaufverfolgung, die die X-Ray Struktur im neuen Stil verwendet, ist in der folgenden Abbildung dargestellt.



Beachten Sie die beiden Segmente im Beispiel. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function` Segment einen Fehler anzeigt, ist bei Ihrer Event-Funktion ein Problem aufgetreten.

In den folgenden Themen finden Sie eine sprachspezifische Einführung in die Ablaufverfolgung in Lambda:

- Node.js Code instrumentieren in AWS Lambda
- Instrumentierung von Python-Code in AWS Lambda
- Instrumentierung von Ruby-Code in AWS Lambda
- Instrumentierung von Java-Code in AWS Lambda
- Go-Code instrumentieren AWS Lambda
- Instrumentierung von C#-Code in AWS Lambda

Eine vollständige Liste der Services, welche die aktive Instrumentierung unterstützen, finden Sie unter Unterstützte AWS-Services im AWS X-Ray -Entwicklerhandbuch.

Standardmäßiges Nachverfolgungsverhalten in Lambda

Wenn die `Active`-Nachverfolgung nicht aktiviert ist, verwendet Lambda standardmäßig den `PassThrough`-Nachverfolgungsmodus.

Im `PassThrough` Modus leitet Lambda den X-Ray Tracing-Header an Downstream-Dienste weiter, sendet jedoch nicht automatisch Traces. Dies gilt auch dann, wenn der Nachverfolgungs-Header eine Entscheidung zum Sampling der Anfrage enthält. Wenn der Upstream-Dienst keinen X-Ray Tracing-Header bereitstellt, generiert Lambda einen Header und trifft die Entscheidung, kein Sampling durchzuführen. Sie können jedoch Ihre eigenen Traces senden, indem Sie Tracing-Bibliotheken aus Ihrem Event-Funktionscode aufrufen.

> **Note**
>
> Bisher sendete Lambda automatisch Nachverfolgungen, wenn Upstream-Services wie Amazon API Gateway einen Nachverfolgungs-Header hinzufügten. Da Lambda Traces nicht automatisch sendet, gibt Ihnen Lambda die Kontrolle, die für Sie wichtigen Event-Funktionen nachzuverfolgen. Wenn Ihre Lösung von diesem passiven Nachverfolgungsverhalten abhängt, wechseln Sie zur `Active`-Nachverfolgung.

Berechtigungen für die Ausführungsrolle

Lambda benötigt die folgenden Berechtigungen, um Trace-Daten an zu X-Ray senden. Fügen Sie sie der Ausführungsrolle Ihrer Event-Funktion hinzu.

- Röntgen: PutTraceSegments
- röntgen: PutTelemetryRecords

Diese Berechtigungen sind in der AWSXRayDaemonWriteAccessverwalteten Richtlinie enthalten.

`Active` Tracing mit der Lambda-API aktivieren

Verwenden Sie die folgenden API-Operationen, um die Ablaufverfolgungskonfiguration mit dem AWS CLI oder AWS SDK zu verwalten:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktivieren von **`Active Tracing`** mit CloudFormation

Verwenden Sie die Eigenschaft, um die Ablaufverfolgung `AWS::Lambda::Function` für eine Ressource in einer CloudFormation Vorlage zu aktivieren. `TracingConfig`

Example function-inline.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` -Ressource die `Tracing` Eigenschaft.

Example template.yml — Konfiguration für die Ablaufverfolgung

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Überwachen Sie die Leistung von Eventfunktionen mit Amazon CloudWatch Lambda Insights

Amazon CloudWatch Lambda Insights sammelt und aggregiert Leistungskennzahlen und Protokolle zur Laufzeit der Lambda Event Function für Ihre serverlosen Anwendungen. Auf dieser Seite wird beschrieben, wie Sie Lambda Insights aktivieren und verwenden, um Probleme mit Ihren Lambda-Ereignisfunktionen zu diagnostizieren.

Sections

- Wie Lambda Insights Serverless-Anwendungen überwacht
- Preisgestaltung
- Unterstützte Laufzeiten
- Lambda Insights in der Lambda-Konsole aktivieren
- Programmgesteuertes Aktivieren von Lambda Insights
- Verwenden des Lambda-Insights-Dashboards
- Beispiel-Workflow zur Erkennung von Anomalien in der Event-Funktion
- Beispiel für einen Workflow, der Abfragen zur Fehlerbehebung bei einer Ereignisfunktion verwendet
- Als nächstes

Wie Lambda Insights Serverless-Anwendungen überwacht

CloudWatch Lambda Insights ist eine Überwachungs- und Fehlerbehebungslösung für serverlose Anwendungen, die auf ausgeführt werden. AWS Lambda Die Lösung erfasst, aggregiert und fasst Metriken auf Systemebene zusammen, einschließlich CPU-Zeit, Arbeitsspeicher, Datenträger- und Netzwerknutzung. Es sammelt, aggregiert und fasst auch Diagnoseinformationen wie Kaltstarts und Abschaltungen von Lambda-Workern zusammen, um Ihnen zu helfen, Probleme mit Ihren Lambda-Ereignisfunktionen zu isolieren und schnell zu lösen.

Lambda Insights verwendet eine neue CloudWatch Lambda Insights-Erweiterung, die als Lambda-Schicht bereitgestellt wird. Wenn Sie diese Erweiterung für eine Lambda-Ereignisfunktion für eine unterstützte Laufzeit aktivieren, sammelt sie Metriken auf Systemebene und gibt für jeden Aufruf dieser Lambda-Ereignisfunktion ein einziges Leistungsprotokollereignis aus. CloudWatch verwendet eine eingebettete Metrikformatierung, um Metriken aus den Protokollereignissen zu extrahieren. Weitere Informationen finden Sie unter AWS Lambda Erweiterungen verwenden.

Die Lambda-Insights-Ebene erweitert die `CreateLogStream` und `PutLogEvents` für die `/aws/lambda-insights/`-Protokollgruppe.

Preisgestaltung

Wenn Sie Lambda Insights für Ihre Lambda Event Function aktivieren, meldet Lambda Insights 8 Metriken pro Event-Funktion und jeder Event-Funktionsaufruf sendet etwa 1 KB Logdaten an. CloudWatch Sie zahlen nur für die von Lambda Insights für Ihre Event Function gemeldeten Metriken und Logs. Es fallen keine Mindestgebühren oder Mindestnutzungsanforderungen an. Sie zahlen nicht für Lambda Insights, wenn die Event Function nicht aufgerufen wird. Ein Preisbeispiel finden Sie unter CloudWatch Amazon-Preise.

Unterstützte Laufzeiten

Sie können Lambda Insights mit jeder Laufzeitumgebung verwenden, die Lambda-Erweiterungen unterstützen.

Lambda Insights in der Lambda-Konsole aktivieren

Sie können die erweiterte Lambda Insights-Überwachung für neue und bestehende Lambda Event Functions aktivieren. Wenn Sie Lambda Insights für eine Event-Funktion in der Lambda-Konsole für eine unterstützte Laufzeit aktivieren, fügt Lambda die Lambda Insights-Erweiterung als Ebene zu Ihrer Event-Funktion hinzu und verifiziert oder versucht, die CloudWatchLambdaInsightsExecutionRolePolicyRichtlinie an die Ausführungsrolle Ihrer Event-Funktion anzuhängen.

Lambda Insights in der Lambda-Konsole aktivieren

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Ihre Event-Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie im linken Menü die Option Überwachungs- und Betriebstools.
5. Wählen Sie im Bereich Zusätzliche Überwachungstools die Option Edit (Bearbeiten).
6. Aktivieren Sie unter CloudWatch Lambda Insights die erweiterte Überwachung.
7. Wählen Sie Save aus.

Programmgesteuertes Aktivieren von Lambda Insights

Sie können Lambda Insights auch mithilfe der CLI AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (SAM) oder der AWS Cloud Development Kit (AWS CDK) aktivieren. CloudFormationWenn Sie Lambda Insights programmgesteuert für eine Event-Funktion für eine unterstützte Laufzeit aktivieren, CloudWatch hängt die CloudWatchLambdaInsightsExecutionRolePolicyRichtlinie der Ausführungsrolle Ihrer Event-Funktion an.

Weitere Informationen finden Sie unter Erste Schritte mit Lambda Insights im CloudWatch Amazon-Benutzerhandbuch.

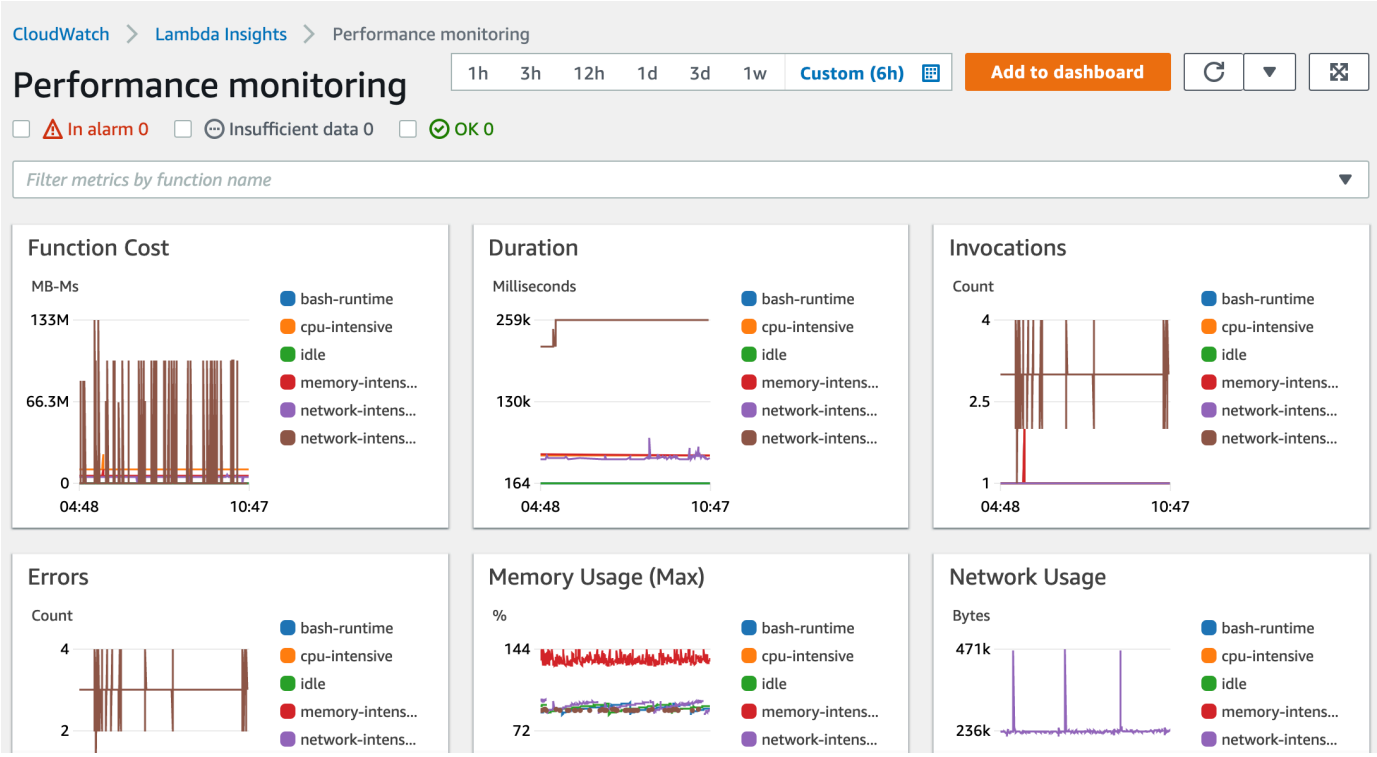
Verwenden des Lambda-Insights-Dashboards

Das Lambda Insights-Dashboard hat zwei Ansichten in der CloudWatch Konsole: die Multifunktionsübersicht und die Einzelfunktionsansicht. In der Multifunktionsübersicht werden die Laufzeitmetriken für die Lambda-Ereignisfunktionen im AWS Girokonto und in der Region zusammengefasst. Die Einzelfunktionsansicht zeigt die verfügbaren Laufzeitmetriken für eine einzelne Lambda-Ereignisfunktion.

Sie können die Multifunktionsübersicht des Lambda Insights-Dashboards in der CloudWatch Konsole verwenden, um zu stark oder zu wenig genutzte Lambda-Ereignisfunktionen zu identifizieren. Sie können die Einzelfunktionsansicht des Lambda Insights-Dashboards in der CloudWatch Konsole verwenden, um einzelne Anfragen zu beheben.

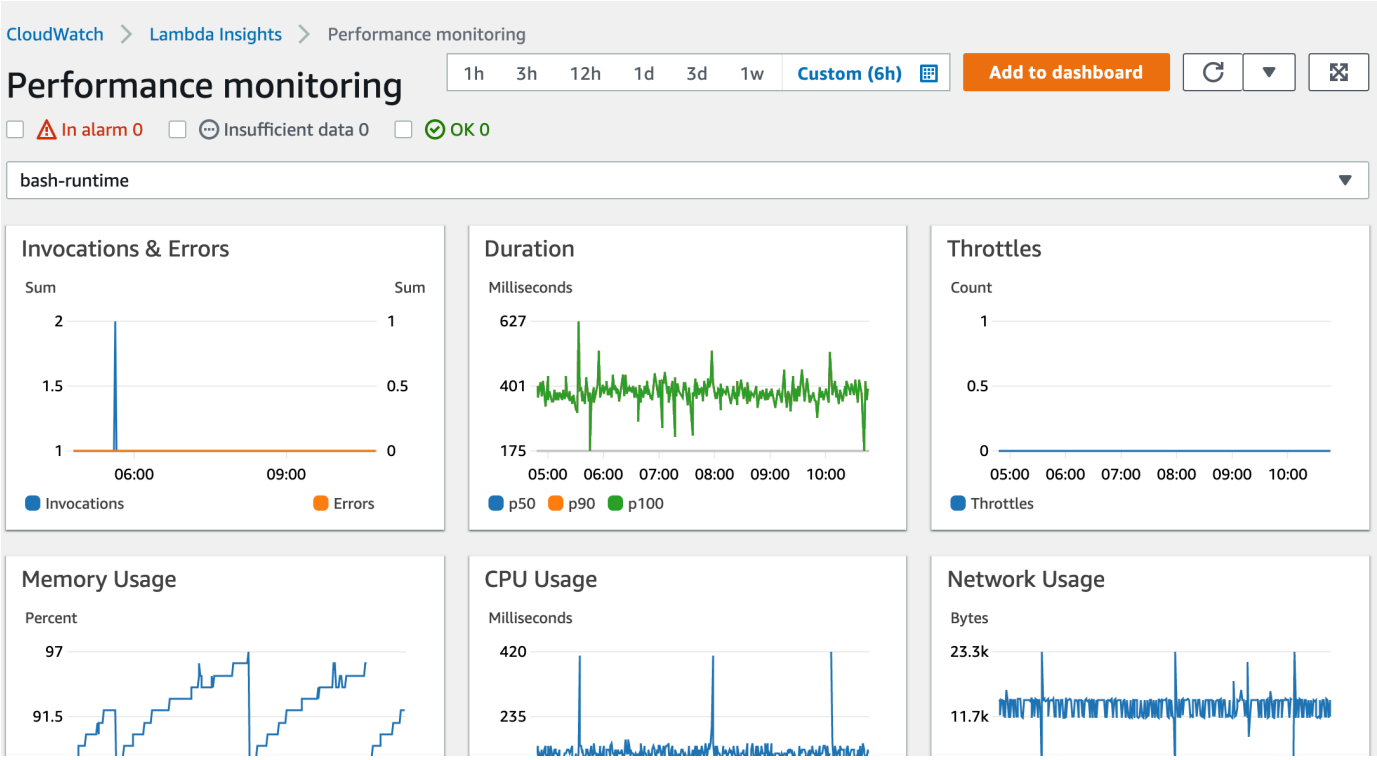
Um die Laufzeitmetriken für alle Event-Funktionen anzuzeigen

1. Öffnen Sie die Multi-functionSeite in der CloudWatch Konsole.
2. Wählen Sie aus den vordefinierten Zeitbereichen oder wählen Sie einen benutzerdefinierten Zeitbereich aus.
3. (Optional) Wählen Sie Zum Dashboard hinzufügen, um die Widgets zu Ihrem CloudWatch Dashboard hinzuzufügen.



Um die Laufzeitmetriken einer einzelnen Event-Funktion anzuzeigen

1. Öffnen Sie die Single-functionSeite in der CloudWatch Konsole.
2. Wählen Sie aus den vordefinierten Zeitbereichen oder wählen Sie einen benutzerdefinierten Zeitbereich aus.
3. (Optional) Wählen Sie Zum Dashboard hinzufügen, um die Widgets zu Ihrem CloudWatch Dashboard hinzuzufügen.



Weitere Informationen finden Sie unter Widgets auf CloudWatch Dashboards erstellen und damit arbeiten.

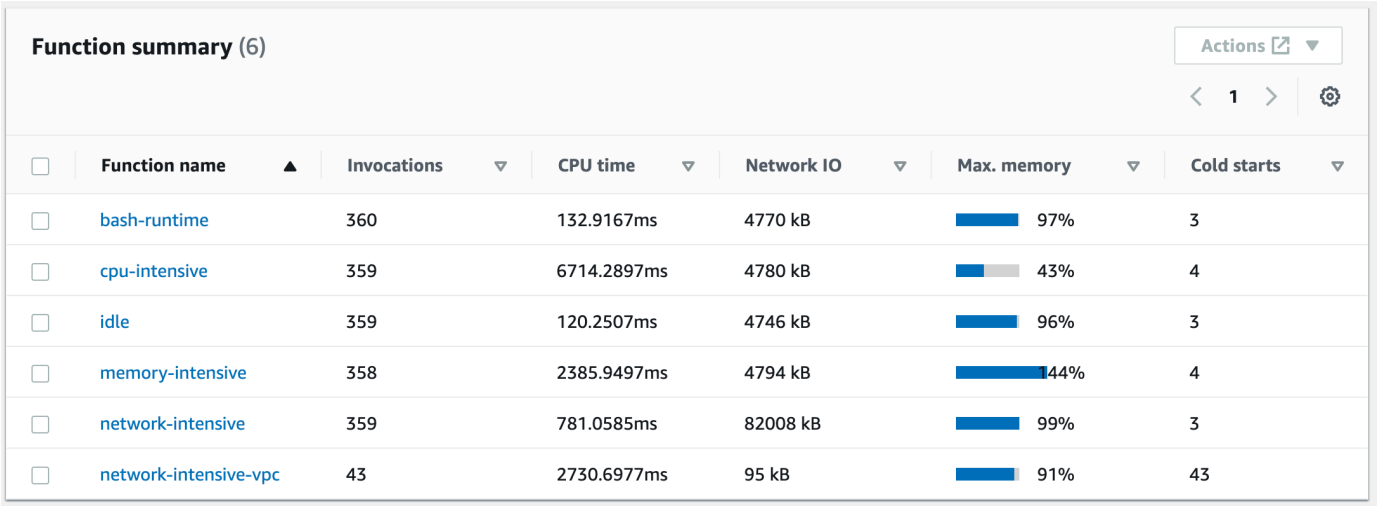
Beispiel-Workflow zur Erkennung von Anomalien in der Event-Funktion

Sie können die Multifunktionsübersicht im Lambda Insights-Dashboard verwenden, um mit Ihrer Event Function Anomalien im Rechenspeicher zu identifizieren und zu erkennen. Wenn die Multifunktionsübersicht beispielsweise darauf hinweist, dass eine Ereignisfunktion eine große Menge an Speicher belegt, können Sie detaillierte Kennzahlen zur Speicherauslastung im Bereich Speicherbelegung einsehen. Sie können dann zum Metrik-Dashboard wechseln, um die Anomalieerkennung zu aktivieren oder einen Alarm zu erstellen.

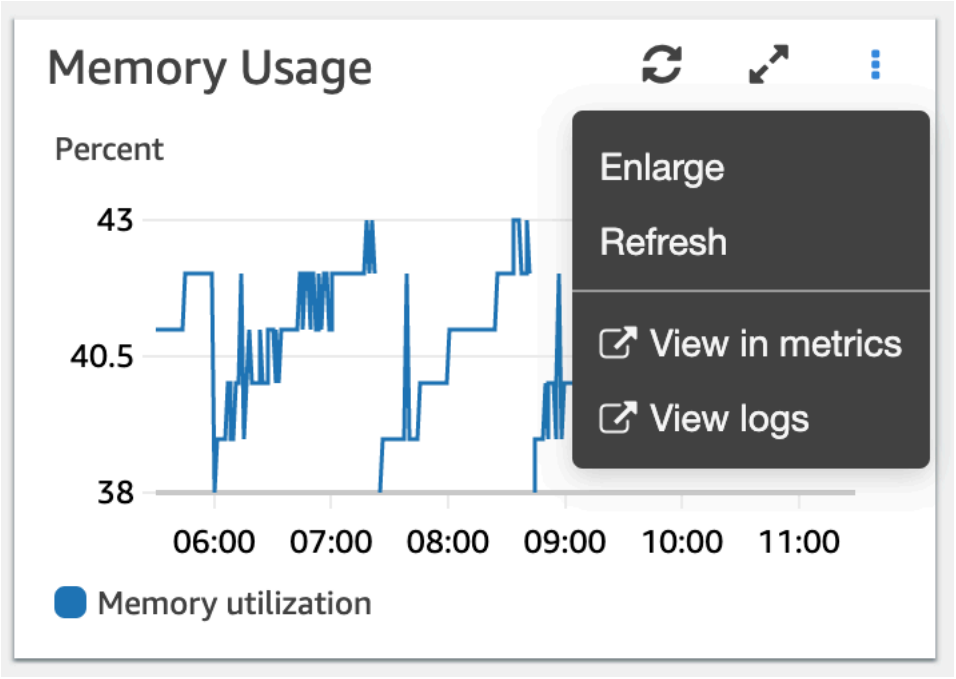
Um die Erkennung von Anomalien für eine Ereignisfunktion zu aktivieren

1. Öffnen Sie die Multi-functionSeite in der CloudWatch Konsole.
2. Wählen Sie unter Funktionsübersicht den Namen Ihrer Event-Funktion aus.

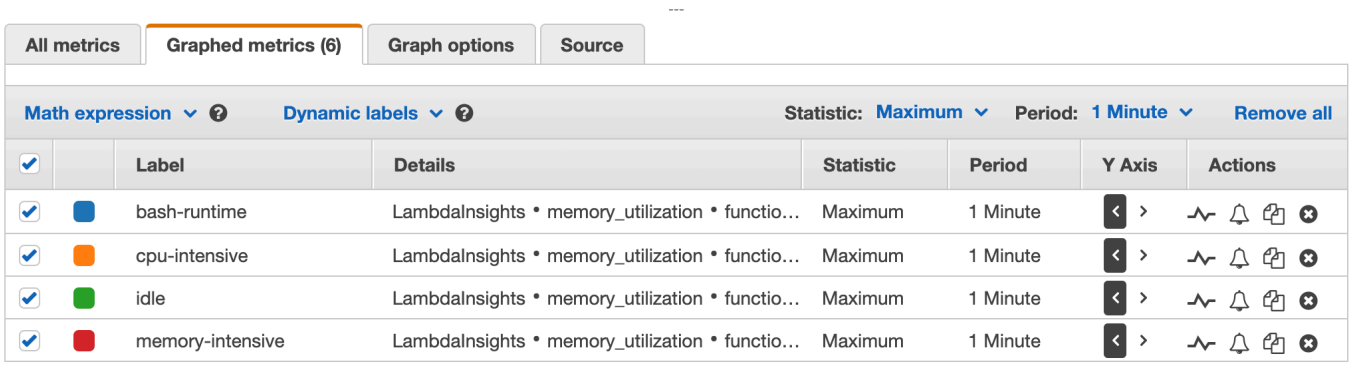
   Die Einzelfunktionsansicht wird mit den Laufzeitmetriken der Event-Funktion geöffnet.



3. Wählen Sie im Bereich Memory Usage (Speicherverwendung) die drei vertikalen Punkte aus, und wählen Sie dann View in metrics (In Metriken anzeigen), um das Metrics (Metriken)-Dashboard zu öffnen.



4. Wählen Sie auf der Registerkarte Graphische Metriken in der Spalte Aktionen das erste Symbol aus, um die Anomalieerkennung für die Event-Funktion zu aktivieren.



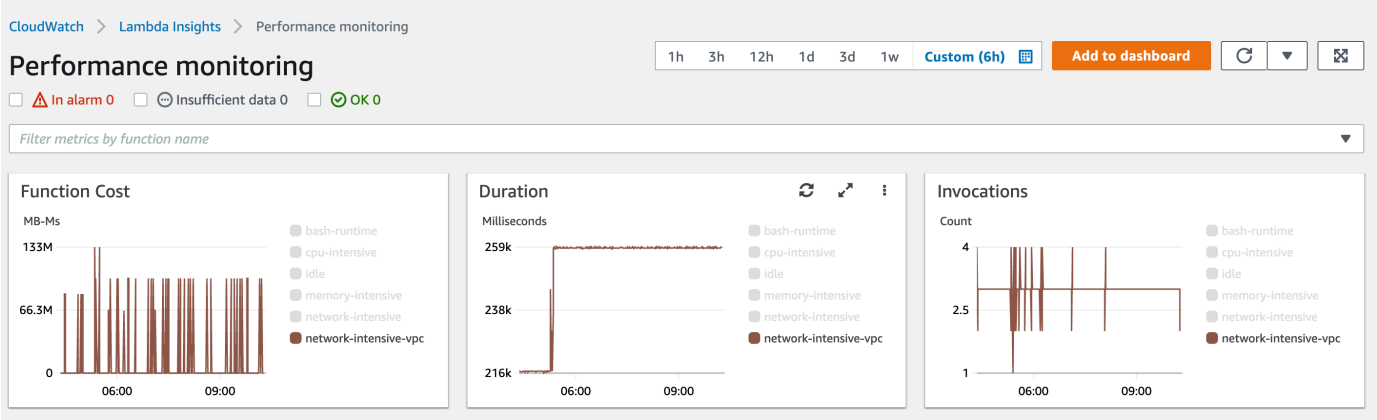
Weitere Informationen finden Sie unter Verwenden der CloudWatch Anomalieerkennung.

Beispiel für einen Workflow, der Abfragen zur Fehlerbehebung bei einer Ereignisfunktion verwendet

Sie können die Einzelfunktionsansicht im Lambda Insights-Dashboard verwenden, um die Hauptursache für einen Anstieg der Dauer von Event-Funktionen zu ermitteln. Wenn in der Multifunktionsübersicht beispielsweise eine starke Zunahme der Dauer der Ereignisfunktion angezeigt wird, können Sie die Dauer der einzelnen Ereignisfunktionen pausieren oder sie im Bereich „Dauer“ auswählen, um festzustellen, welche Ereignisfunktion die Erhöhung verursacht. Sie können dann zur Einzelfunktionsansicht wechseln und die Anwendungsprotokolle überprüfen, um die Ursache zu ermitteln.

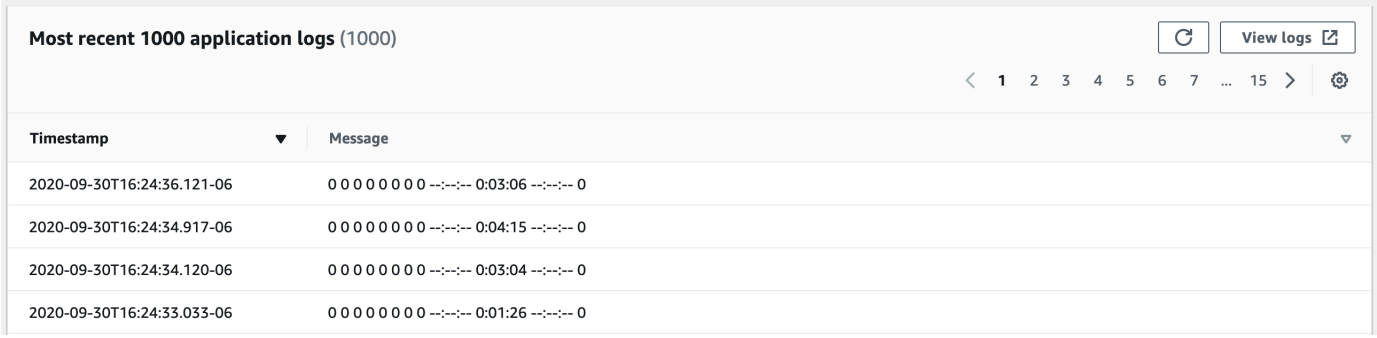
So führen Sie Abfragen für eine Ereignisfunktion aus

1. Öffnen Sie die Multi-functionSeite in der CloudWatch Konsole.
2. Wählen Sie im Bereich Duration (Dauer) Ihre Funktion aus, um die Dauer-Metriken zu filtern.

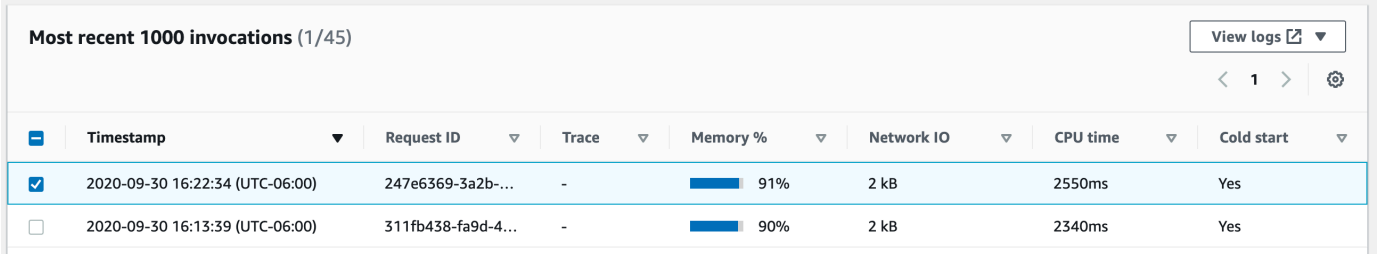


3. Öffne die Single-functionSeite.

4. Wählen Sie die Dropdown-Liste Filter metrics by function name (Metriken nach Funktionsnamen filtern) und wählen Sie dann Ihre Funktion aus.
5. Um die neuesten 1000 Anwendungsprotokolle anzuzeigen, wählen Sie die Registerkarte Application logs (Anwendungsprotokolle).
6. Überprüfen Sie den Zeitstempel und die Meldung um die Aufrufanforderung zu identifizieren, für die Sie Fehler beheben möchten.

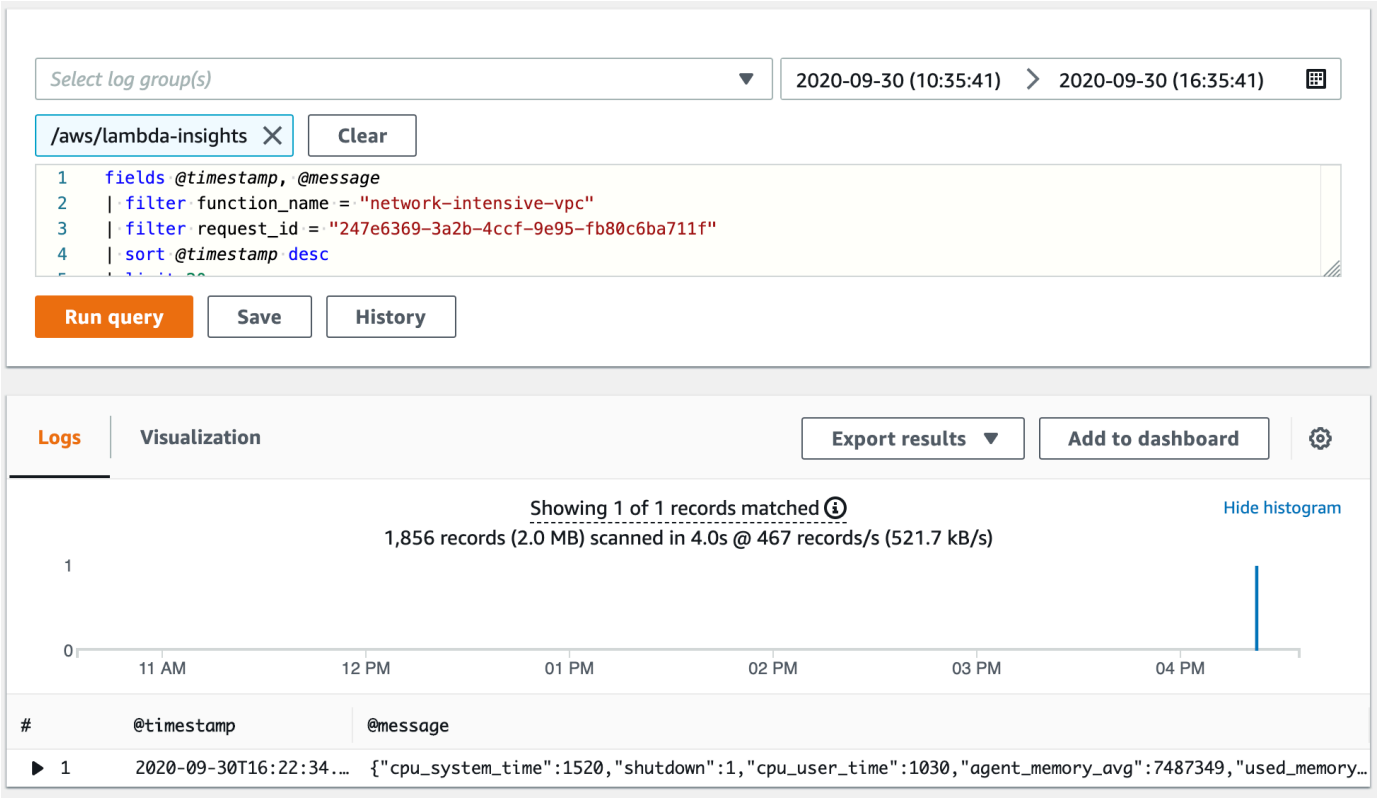


7. Um die letzten 1000 Aufrufe anzuzeigen, wählen Sie die Registerkarte Invocations (Aufrufe).
8. Wählen Sie den Zeitstempel oder die Meldung für die Aufrufanforderung aus, für die Sie Fehler beheben möchten.



9. Wählen Sie die Dropdown-Liste View logs (Protokolle anzeigen) und wählen Sie dann View performance logs (Leistungsprotokolle anzeigen) aus.

   Eine automatisch generierte Abfrage für Ihre Event-Funktion wird im Logs Insights-Dashboard geöffnet.
10. Wählen Sie Run query (Abfrage ausführen), um eine Logs (Protokolle)-Meldung für die Aufrufanforderung zu generieren.



Als nächstes

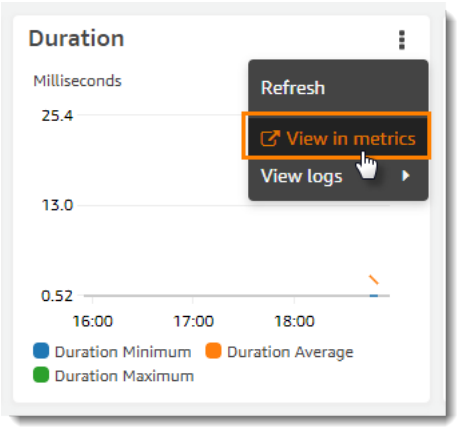
- Weitere Informationen zum Erstellen eines CloudWatch Logs-Dashboards finden Sie unter Create a Dashboard im CloudWatch Amazon-Benutzerhandbuch.
- Weitere Informationen zum Hinzufügen von Abfragen zu einem CloudWatch Logs-Dashboard finden Sie unter „Abfrage zum Dashboard hinzufügen“ oder „Abfrageergebnisse exportieren“ im CloudWatch Amazon-Benutzerhandbuch.

Überwachen von Lambda-Anwendungen

Der Bereich „Anwendungen“ der Lambda-Konsole enthält eine Registerkarte „Überwachung“, auf der Sie ein CloudWatch Amazon-Dashboard mit aggregierten Kennzahlen für die Ressourcen in Ihrer Anwendung überprüfen können.

So überwachen Sie eine Lambda-Anwendung

1. Öffnen Sie die Seite Anwendungen der Lambda-Konsole.
2. Wählen Sie Monitoring.
3. Um weitere Details zu den Metriken in einem beliebigen Diagramm anzuzeigen, wählen Sie im Dropdown-Menü Ansicht in Metriken.



   Das Diagramm erscheint in einer neuen Registerkarte, wobei die relevanten Metriken unterhalb des Diagramms aufgelistet sind. Sie können Ihre Ansicht dieses Diagramms anpassen, indem Sie die angezeigten Metriken und Ressourcen, die Statistik, den Zeitraum und andere Faktoren ändern, um ein besseres Verständnis der aktuellen Situation zu erhalten.

Standardmäßig zeigt die Lambda-Konsole ein grundlegendes Dashboard an. Sie können diese Seite anpassen, indem Sie Ihrer Anwendungsvorlage ein oder mehrere CloudWatch Amazon-Dashboards mit dem Ressourcentyp AWS::CloudWatch: :Dashboard hinzufügen. Wenn Ihre Vorlage ein oder mehrere Dashboards enthält, zeigt die Seite Ihre Dashboards anstelle des Standard-Dashboards an. Sie mit dem Dropdown-Menü oben rechts auf der Seite zwischen Dashboards wechseln. Im folgenden Beispiel wird ein Dashboard mit einem einzigen Widget erstellt, das die Anzahl der Aufrufe einer Ereignisfunktion mit dem Namen grafisch darstellt. `my-function`

Example Dashboard-Vorlage für eine Event-Funktion

```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
                                    "label": "Total Invocations"
                                }
                            ]
                        ],
                        "region": "us-east-1",
                        "title": "Invocations",
                        "view": "timeSeries",
                        "stacked": false
                    }
                }
            ]
        }
```

Weitere Informationen zum Erstellen von CloudWatch Dashboards und Widgets finden Sie unter Struktur und Syntax des Dashboards in der Amazon CloudWatch API-Referenz.

Überwachen Sie die Anwendungsleistung mit Amazon CloudWatch Application Signals

Amazon CloudWatch Application Signals ist eine APM-Lösung (Application Performance Monitoring), mit der Entwickler und Betreiber den Zustand und die Leistung ihrer mit Lambda erstellten serverlosen Anwendungen überwachen können. Sie können Application Signals mit einem Klick von der Lambda-Konsole aus aktivieren, und Sie müssen Ihrer Lambda-Ereignisfunktion keinen Instrumentierungscode oder externe Abhängigkeiten hinzufügen. Nachdem Sie Application Signals aktiviert haben, können Sie alle gesammelten Metriken und Traces in der Konsole anzeigen. CloudWatch Auf dieser Seite wird beschrieben, wie Sie Application-Signals-Telemetriedaten für Ihre Anwendungen aktivieren und anzeigen.

Themen

- So lässt sich Application Signals in Lambda integrieren
- Preisgestaltung
- Unterstützte Laufzeiten
- Aktivieren von Application Signals in der Lambda-Konsole
- Verwenden des Application Signals-Dashboards

So lässt sich Application Signals in Lambda integrieren

Application Signals instrumentiert Ihre Lambda-Ereignisfunktionen automatisch mithilfe erweiterter AWS Distro for OpenTelemetry (ADOT) -Bibliotheken, die über eine Lambda-Schicht bereitgestellt werden. Application Signals liest die von der Ebene gesammelten Daten und generiert Dashboards mit wichtigen Leistungsmetriken für Ihre Anwendungen.

Sie können diese Ebene mit einem Klick anhängen, indem Sie Application Signals in der Lambda-Konsole aktivieren. Wenn Sie Application Signals über die Konsole aktivieren, führt Lambda Folgendes in Ihrem Namen aus:

- Aktualisiert die Ausführungsrolle Ihrer Event-Funktion um die. `CloudWatchLambdaApplicationSignalsExecutionRolePolicy` Diese Richtlinie bietet Schreibzugriff auf AWS X-Ray und CloudWatch protokolliert Gruppen, die für Application Signals verwendet werden.
- Fügt Ihrer Event-Funktion eine Ebene hinzu, die die Event-Funktion automatisch instrumentiert, um Telemetriedaten wie Anfragen, Verfügbarkeit, Latenz, Fehler und Störungen zu erfassen. Um

sicherzustellen, dass Application Signals ordnungsgemäß funktioniert, entfernen Sie jeglichen vorhandenen X-Ray SDK-Instrumentierungscode aus Ihrer Event-Funktion. Benutzerdefinierter X-Ray SDK-Instrumentierungscode kann die vom Layer bereitgestellte Instrumentierung beeinträchtigen.

- Fügt die `AWS_LAMBDA_EXEC_WRAPPER` Umgebungsvariable zu Ihrer Event-Funktion hinzu und setzt ihren Wert auf. `/opt/otel-instrument` Diese Umgebungsvariable ändert das Startverhalten Ihrer Event-Funktion, sodass sie den Layer Application Signals verwendet. Sie ist für die korrekte Instrumentierung erforderlich. Wenn diese Umgebungsvariable bereits existiert, stellen Sie sicher, dass sie auf den erforderlichen Wert gesetzt ist.

Preisgestaltung

Die Verwendung von Anwendungssignalen für Ihre Lambda-Ereignisfunktionen verursacht Kosten. Preisinformationen finden Sie unter CloudWatch Amazon-Preise.

Unterstützte Laufzeiten

Die Application Signal-Integration mit Lambda funktioniert mit den folgenden Laufzeiten:

- .NET 8
- Java 11
- Java 17
- Java 21
- Python 3.10
- Python 3.11
- Python 3.12
- Python 3.13
- Node.js 18.x
- Node.js 20,x
- Node.js 22,x

Aktivieren von Application Signals in der Lambda-Konsole

Sie können Application Signals für jede vorhandene Lambda-Ereignisfunktion mithilfe einer unterstützten Laufzeit aktivieren. In den folgenden Schritten wird beschrieben, wie Sie Application Signals mit einem Klick in der Lambda-Konsole aktivieren.

So aktivieren Sie Application Signals in der Lambda-Konsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Ihre Event-Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie im linken Menü die Option Überwachungs- und Betriebstools.
5. Wählen Sie im Bereich Zusätzliche Überwachungstools die Option Edit (Bearbeiten).
6. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray unter Anwendungssignale die Option Aktivieren aus.
7. Wählen Sie Speichern.

Wenn Sie Application Signals zum ersten Mal für Ihre Event Function aktivieren, müssen Sie außerdem eine einmalige Serviceerkennung für Application Signals in der CloudWatch Konsole einrichten. Nachdem Sie dieses einmalige Service Discovery-Setup abgeschlossen haben, erkennt Application Signals automatisch alle zusätzlichen Lambda-Ereignisfunktionen, für die Sie Application Signals aktivieren, in allen Regionen.

> **Note**
>
> Nachdem Sie Ihre aktualisierte Event Function aufgerufen haben, kann es bis zu 10 Minuten dauern, bis Servicedaten im Application Signals Dashboard in der Konsole angezeigt werden. CloudWatch

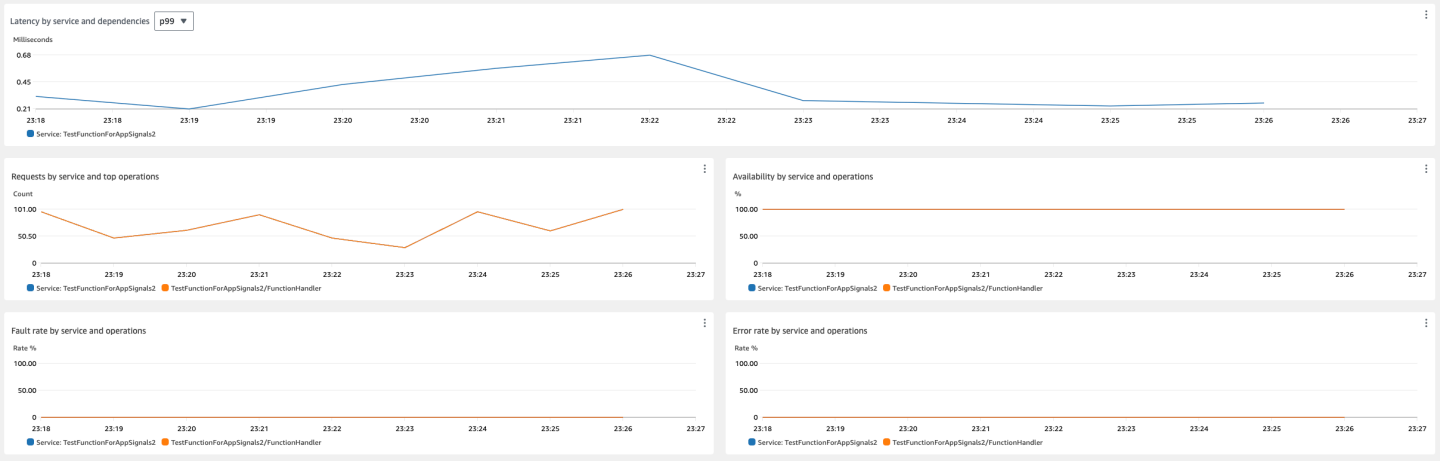
Verwenden des Application Signals-Dashboards

Nachdem Sie Application Signals für Ihre Event Function aktiviert haben, können Sie Ihre Anwendungsmetriken in der CloudWatch Konsole visualisieren. Mit den folgenden Schritten können Sie das zugehörige Application-Signals-Dashboard schnell von der Lambda-Konsole aus aufrufen:

Um das Application Signals-Dashboard für Ihre Event-Funktion anzuzeigen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Ihre Event-Funktion aus.
3. Wählen Sie den Tab Überwachung.
4. Wählen Sie die Schaltfläche Anwendungssignale anzeigen. Dadurch gelangen Sie direkt zur Übersicht der Anwendungssignale für Ihren Service in der CloudWatch Konsole.

Der folgende Screenshot zeigt beispielsweise Messwerte für Latenz, Anzahl der Anfragen, Verfügbarkeit, Fehlerrate und Fehlerrate für eine Event-Funktion in einem Zeitfenster von 10 Minuten.



Um das Beste aus Ihrer Integration mit Application Signals herauszuholen, können Sie Service Level Objectives (SLOs) für Ihre Anwendung erstellen. Sie können beispielsweise Latenz-SLOs erstellen, um sicherzustellen, dass Ihre Anwendung schnell auf Benutzeranfragen reagiert, und Verfügbarkeits-SLOs, um die Verfügbarkeit zu verfolgen. SLOs können Ihnen helfen, Leistungseinbußen oder Ausfälle zu erkennen, bevor sie sich auf Ihre Benutzer auswirken. Weitere Informationen finden Sie unter Service Level Objectives (SLOs) im CloudWatch Amazon-Benutzerhandbuch.

Ferndebuggen von Lambda-Ereignisfunktionen mit Visual Studio Code

Mit der Remote-Debugging-Funktion in der können Sie Ihre Lambda-Ereignisfunktionen debuggen AWS Toolkit for Visual Studio Code, die direkt in der Cloud ausgeführt werden. AWS Dies ist nützlich, wenn Sie Probleme untersuchen, die lokal schwer zu reproduzieren sind oder nur anhand von Protokollen diagnostiziert werden können.

Mit Remote-Debugging können Sie:

- Legen Sie Breakpoints in Ihrem Lambda Event Function Code fest.
- Die Codeausführung in Echtzeit durchlaufen.
- Variablen und Status während der Laufzeit überprüfen.
- Debuggen Sie Lambda-Ereignisfunktionen AWS, die bereitgestellt werden, einschließlich solcher in VPCs oder mit bestimmten IAM-Berechtigungen.

Unterstützte Laufzeiten

Remote-Debugging wird für die folgenden Laufzeiten unterstützt:

- Python (AL2023)
- Java
- JavaScript/Node.js (AL2023)

> **Note**
>
> Remote-Debugging wird sowohl für x86_64- als auch für arm64-Architekturen unterstützt.

Sicherheit und Remote-Debugging

Das Remote-Debugging erfolgt innerhalb der bestehenden Lambda-Sicherheitsgrenzen. Benutzer können Ebenen an eine Ereignisfunktion anhängen, indem sie die `UpdateFunctionConfiguration` Berechtigung verwenden, die bereits Zugriff auf Umgebungsvariablen und die Konfiguration von Event Function hat. Das Remote-Debugging geht nicht über diese bestehenden Berechtigungen hinaus. Stattdessen werden zusätzliche Sicherheitskontrollen durch sicheres Tunneling und automatische Sitzungsverwaltung hinzugefügt.

Darüber hinaus ist das Remote-Debugging ein vollständig vom Kunden kontrolliertes Feature, das explizite Berechtigungen und Aktionen erfordert:

- Erstellung eines sicheren IoT-Tunnels: Das AWS Toolkit muss einen sicheren IoT-Tunnel erstellen. Dies erfolgt nur mit ausdrücklicher Genehmigung des Benutzers. `iot:OpenTunnel`
- Anhängen der Debugging-Ebene und Tokenverwaltung: Der Debugging-Prozess gewährleistet die Sicherheit durch folgende Kontrollen:
  - Die Debugging-Schicht muss an die Lambda-Ereignisfunktion angehängt werden, und für diesen Vorgang sind die folgenden Berechtigungen erforderlich: `lambda:UpdateFunctionConfiguration` und. `lambda:GetLayerVersion`
  - Aktualisieren Sie das Sicherheitstoken (generiert durch`iot:OpenTunnel`) in der Umgebungsvariablen Event Function vor jeder Debug-Sitzung. Dieser Schritt erfordert `lambda:UpdateFunctionConfiguration` auch.
  - Aus Sicherheitsgründen wird dieses Token automatisch rotiert und die Debug-Ebene wird am Ende jeder Debugging-Sitzung automatisch entfernt und kann nicht wiederverwendet werden.

> **Note**
>
> Remote-Debugging wird sowohl für x86_64- als auch für arm64-Architekturen unterstützt.

Voraussetzungen

Bevor Sie mit dem Remote-Debugging beginnen, stellen Sie sicher, dass Sie über Folgendes verfügen:

1. Eine Lambda-Ereignisfunktion, die für Ihr AWS Konto bereitgestellt wurde.
2. AWS Toolkit for Visual Studio Code. Installationsanweisungen finden Sie unter Einrichten des AWS Toolkit for Visual Studio Code.
3. Die Version des AWS Toolkits, die Sie installiert haben, ist 3.69.0 oder höher.
4. AWS Die Anmeldeinformationen sind in konfiguriert. AWS Toolkit for Visual Studio Code Weitere Informationen finden Sie unter Authentifizierung und Zugriffskontrolle.

Lambda-Ereignisfunktionen aus der Ferne debuggen

Führen Sie die folgenden Schritte aus, um eine Remote-Debugging-Sitzung zu starten:

1. Öffnen Sie den AWS Explorer in VS Code, indem Sie AWS das Symbol in der linken Seitenleiste auswählen.
2. Erweitern Sie den Abschnitt Lambda, um Ihre Event-Funktionen zu sehen.
3. Öffnen Sie das Kontextmenü für die Event-Funktion, die Sie debuggen möchten.
4. Wählen Sie im Kontextmenü die Option Remote aufrufen aus.
5. Aktivieren Sie im sich öffnenden Aufruffenster das Kontrollkästchen Debugging aktivieren.
6. Klicken Sie auf Aufrufen, um die Remote-Debugging-Sitzung zu starten.

> **Note**
>
> Lambda-Ereignisfunktionen haben ein kombiniertes Limit von 250 MB für den Ereignisfunktionscode und alle angehängten Ebenen. Die Remote-Debugging-Schicht erhöht die Größe Ihrer Event-Funktion um etwa 40 MB.

Eine Remote-Debugging-Sitzung endet, wenn Sie:

- auf dem Bildschirm mit der Remoteaufruf-Konfiguration“ die Option Debug-Setup entfernen auswählen
- in den Debugging-Steuerelementen von VS Code das Symbol „Trennen“ auswählen
- die Handler-Datei im VS-Code-Editor auswählen.

> **Note**
>
> Die Debugging-Ebene wird automatisch entfernt, wenn nach Ihrem letzten Aufruf 60 Sekunden lang keine Aktivität stattfindet.

Deaktivieren von Remote-Debugging

Es gibt drei Möglichkeiten, dieses Feature zu deaktivieren:

- Funktionsupdates verweigern: `lambda:UpdateFunctionConfiguration` auf `deny` setzen
- IoT-Berechtigungen einschränken: IoT-related Berechtigungen verweigern
- Debug-Ebenen blockieren: `lambda:GetLayerVersion` für die folgenden ARNs verweigern:

- `arn:aws:lambda:*:*:layer:LDKLayerX86:*`
- `arn:aws:lambda:*:*:layer:LDKLayerArm64:*`

> **Note**
>
> Durch die Deaktivierung dieser Funktion wird verhindert, dass die Debugging-Ebene bei Aktualisierungen der Konfiguration der Event-Funktion hinzugefügt wird.

Zusätzliche Informationen

Weitere Informationen zur Verwendung von Lambda in VS Code finden Sie unter Lokale Entwicklung von Lambda-Funktionen mit VS Code.

Ausführliche Anweisungen zur Fehlerbehebung, zu erweiterten Anwendungsfällen und zur regionalen Verfügbarkeit finden Sie unter Lambda-Funktionen per Fernzugriff im AWS Toolkit for Visual Studio Code Benutzerhandbuch.

Lambda mit Ereignissen von anderen aufrufen AWS service

Einige AWS-Services können Lambda-Ereignisfunktionen mithilfe von Triggern direkt aufrufen. Diese Dienste leiten Ereignisse an Lambda weiter und die Funktion wird sofort aufgerufen, wenn das angegebene Ereignis eintritt. Trigger eignen sich für diskrete Ereignisse und die Verarbeitung in Echtzeit. Wenn Sie mit der Lambda-Konsole einen Trigger erstellen, interagiert die Konsole mit dem entsprechenden AWS Dienst, um die Ereignisbenachrichtigung für diesen Dienst zu konfigurieren. Der Auslöser wird tatsächlich von dem Dienst gespeichert und verwaltet, der die Ereignisse generiert, nicht von Lambda.

Die Ereignisse sind im JSON-Format gegliedert. Die JSON-Struktur variiert je nach Service, der sie generiert und dem Ereignistyp, aber sie alle enthalten die Daten, die die Funktion benötigt, um das Ereignis zu verarbeiten.

Zusätzlich zu ereignisgesteuerten Aufrufen können Sie Lambda mit Amazon EC2, selbstverwaltetem Apache Kafka und Kubernetes verwenden.

Eine Event-Funktion kann mehrere Auslöser haben. Jeder Auslöser fungiert als Client, der Ihre Funktion unabhängig aufruft, und jedes Ereignis, das Lambda an Ihre Funktion weitergibt, enthält Daten von nur einem Auslöser. Lambda wandelt das Ereignisdokument in ein Objekt um und leitet es an Ihren Funktions-Handler weiter.

Je nach Service kann der ereignisgesteuerte Aufruf synchron oder asynchron erfolgen.

- Beim synchronen Aufruf wartet der Service, der das Ereignis generiert, auf die Antwort Ihrer Funktion. Dieser Service definiert die Daten, die die Funktion in der Antwort zurückgeben muss. Der Service steuert die Fehlerstrategie, z. B. ob bei Fehlern ein erneuter Versuch unternommen werden soll.
- Bei asynchronen Aufrufen verschiebt Lambda das Ereignis in die Warteschlange, bevor sie es an Ihre Funktion übergibt. Wenn Lambda das Ereignis in die Warteschlange stellt, sendet es sofort eine Erfolgsantwort an den Service der das Ereignis generiert hat. Nachdem die Funktion das Ereignis verarbeitet hat, gibt Lambda keine Antwort an den ereignisgenerierenden Dienst zurück.

Erstellen eines Auslösers

Der einfachste Weg, einen Auslöser zu erstellen, ist die Verwendung der Lambda-Konsole. Wenn Sie mit der Konsole einen Trigger erstellen, fügt Lambda der ressourcenbasierten Richtlinie der Event-Funktion automatisch die erforderlichen Berechtigungen hinzu.

So erstellen Sie einen Auslöser mit der Lambda-Konsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie einen Auslöser erstellen möchten.
3. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).
4. Wählen Sie den AWS Dienst aus, für den Sie Ihre Funktion aufrufen möchten.
5. Füllen Sie die Optionen im Bereich Trigger-Konfiguration aus und wählen Sie Hinzufügen. Je nachdem, wie AWS-Service Sie Ihre Funktion aufrufen möchten, unterscheiden sich die Konfigurationsoptionen für den Trigger.

Dienste, die Lambda-Ereignisfunktionen aufrufen können

In der folgenden Tabelle sind Dienste aufgeführt, die Lambda-Ereignisfunktionen aufrufen können.

Service	Methode des Aufrufs
Amazon Managed Streaming für Apache Kafka	Zuordnung von Ereignisquellen

Service	Methode des Aufrufs
Self-managed Apache Kafka	Zuordnung von Ereignisquellen
Amazon API Gateway	Event-driven; synchroner Aufruf
AWS CloudFormation	Event-driven; asynchroner Aufruf
CloudWatch Amazon-Protokolle	Event-driven; asynchroner Aufruf
AWS CodeCommit	Event-driven; asynchroner Aufruf
AWS CodePipeline	Event-driven; asynchroner Aufruf
Amazon Cognito	Event-driven; synchroner Aufruf
AWS Config	Event-driven; asynchroner Aufruf
Kunde Connect	Event-driven; synchroner Aufruf
Amazon DocumentDB	Zuordnung von Ereignisquellen
Amazon-DynamoDB	Zuordnung von Ereignisquellen
Elastic Load Balancing (Application Load Balancer)	Event-driven; synchroner Aufruf
EventBridge	Event-driven; asynchroner Aufruf (Ereignisbusse und Regeln), synchroner oder asynchroner Aufruf (Pipes)
Amazon EventBridge Scheduler	Event-driven (zeitbasiert); asynchroner Aufruf
AWS IoT	Event-driven; asynchroner Aufruf
Amazon Kinesis	Zuordnung von Ereignisquellen
Amazon Data Firehose	Event-driven; synchroner Aufruf
Amazon Lex	Event-driven; synchroner Aufruf

Service	Methode des Aufrufs
Amazon MQ	Zuordnung von Ereignisquellen
Amazon Simple Email Service	Event-driven; asynchroner Aufruf
Amazon Simple Notification Service	Event-driven; asynchroner Aufruf
Amazon Simple Queue Service	Zuordnung von Ereignisquellen
Amazon Simple Storage Service (Amazon-S3)	Event-driven; asynchroner Aufruf
Amazon Simple Storage Service Batch	Event-driven; synchroner Aufruf
Secrets Manager	Secret-Rotation
AWS Step Functions	Event-driven; synchroner oder asynchroner Aufruf
Amazon VPC Lattice	Event-driven; synchroner Aufruf

Verwendung von Lambda mit Apache Kafka

Lambda unterstützt Apache Kafka als Ereignisquelle. Apache Kafka ist eine Open-Source-Plattform für Ereignisstreaming, die für die Verarbeitung von Datenpipelines und Streaminganwendungen mit hohem Durchsatz in Echtzeit entwickelt wurde. Es gibt zwei Hauptmethoden für die Verwendung von Lambda mit Apache Kafka:

- the section called “MSK” — Amazon Managed Streaming für Apache Kafka (Amazon MSK) ist ein vollständig verwalteter Dienst von AWS. Amazon MSK unterstützt die Automatisierung der Verwaltung Ihrer Kafka-Infrastruktur, einschließlich Bereitstellung, Patching und Skalierung.
- the section called “Self-managed Apache Kafka” — Eins AWS Terminologie: Ein selbstverwalteter Cluster umfasst nicht-AWS gehostete Kafka-Cluster. Sie können Lambda beispielsweise weiterhin mit einem Kafka-Cluster verwenden, der von einem anderen Anbieter gehostet wird AWS Cloud-Anbieter wie Confluent Cloud oder Roter Panda.

Bei der Entscheidung zwischen Amazon MSK und selbstverwaltetem Apache Kafka sollten Sie Ihre betrieblichen Anforderungen und Kontrollanforderungen berücksichtigen. Amazon MSK ist eine bessere Wahl, wenn Sie möchten AWS um Ihnen schnell bei der Verwaltung eines skalierbaren, produktionsbereiten Kafka-Setups mit minimalem Betriebsaufwand zu helfen. Es vereinfacht die Sicherheit, Überwachung und Hochverfügbarkeit und hilft Ihnen, sich auf die Anwendungsentwicklung statt auf die Infrastrukturverwaltung zu konzentrieren. Auf der anderen Seite eignet sich das selbstverwaltete Apache Kafka besser für Anwendungsfälle, die auf anderen Geräten laufen AWS gehostete Umgebungen, einschließlich lokaler Cluster.

Themen

- Verwenden von Lambda mit Amazon MSK
- Verwendung von Lambda mit selbstverwaltetem Apache Kafka
- Skalierungsmodi des Ereignis-Pollers von Apache Kafka in Lambda
- Apache-Kafka-Abfrage und Stream-Startpositionen in Lambda
- Abfragen von Kafka-Themen im Warteschlangenmodus
- Anpassbare Verbrauchergruppen-ID in Lambda
- Filtern von Ereignissen aus Amazon MSK und selbstverwalteten Apache-Kafka-Ereignisquellen
- Verwenden von Schemaregistern mit Kafka-Ereignisquellen in Lambda
- Verarbeitung mit niedriger Latenz für Kafka-Ereignisquellen
- Konfiguration der Fehlerbehandlungssteuerungen für Kafka-Ereignisquellen

- Erfassen verworfener Batches für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen
- Ein Kafka-Thema als Ziel für den Fall eines Fehlers verwenden
- Protokollierung der Zuordnung von Kafka-Ereignisquellen
- Fehlerbehebung bei der Kafka-Zuordnung von Ereignisquellen

Verwenden von Lambda mit Amazon MSK

Amazon Managed Streaming for Apache Kafka (Amazon MSK) ist ein vollständig verwalteter Service, mit dem Sie Anwendungen erstellen und ausführen können, die Apache Kafka zum Verarbeiten von Streamingdaten verwenden. Amazon MSK vereinfacht die Einrichtung, Skalierung und Verwaltung von Kafka-Clustern. Amazon MSK erleichtert auch die Konfiguration Ihrer Anwendung für mehrere Availability Zones und aus Sicherheitsgründen mit AWS Identity and Access Management (IAM).

In diesem Kapitel wird erklärt, wie Sie einen Amazon MSK-Cluster als Ereignisquelle für Ihre Lambda Event Function verwenden. Der allgemeine Prozess zur Integration von Amazon MSK mit Lambda umfasst die folgenden Schritte:

1. Cluster- und Netzwerk-Setup: Richten Sie zunächst Ihren Amazon-MSK-Cluster ein. Dazu gehört die richtige Netzwerkkonfiguration, damit Lambda auf Ihren Cluster zugreifen kann.
2. Einrichtung der Zuordnung von Ereignisquellen: Erstellen Sie anschließend die Ressource für die Zuordnung von Ereignisquellen, die Lambda benötigt, um Ihren Amazon-MSK-Cluster sicher mit Ihrer Funktion zu verbinden.
3. Einrichtung der Ereignisfunktion und der Berechtigungen — Stellen Sie abschließend sicher, dass Ihre Funktion korrekt eingerichtet ist und über die erforderlichen Berechtigungen für die Ausführung verfügt.

> **Note**
>
> Sie können jetzt Ihre Amazon MSK-Ereignisquellenzuordnungen direkt von der Lambda- oder der Amazon MSK-Konsole aus erstellen und verwalten. Beide Konsolen bieten die Option, die Einrichtung der erforderlichen Berechtigungen für die Lambda-Ausführungsrolle automatisch vorzunehmen, um den Konfigurationsprozess zu optimieren.

Beispiele für die Einrichtung einer Lambda-Integration mit einem Amazon MSK-Cluster finden Sie unter the section called “Tutorial” Verwenden von Amazon MSK als Ereignisquelle für AWS Lambda im AWS Compute-Blog und Amazon MSK Lambda-Integration in den Amazon MSK Labs.

Themen

- Beispielereignis
- Konfiguration Ihres Amazon-MSK-Clusters und Ihres Amazon-VPC-Netzwerks für Lambda
- Konfiguration von Lambda-Berechtigungen für Amazon MSK-Ereignisquellenzuordnungen

- Konfigurieren von Amazon MSK-Ereignisquellen für Lambda
- Tutorial: Verwenden einer Amazon MSK-Ereignisquellenzuordnung zum Aufrufen einer Lambda-Ereignisfunktion

Beispielereignis

Lambda sendet den Nachrichtenstapel im Event-Parameter, wenn es Ihre Event-Funktion aufruft. Die Ereignisnutzlast enthält ein Array von Meldungen. Jedes Array-Element enthält Details zum Amazon-MSK-Thema und zur Partitions-ID sowie einen Zeitstempel und eine base64-codierte Nachricht.

```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
```

```
                }
            ]
        }
    ]
  }
}
```

Konfiguration Ihres Amazon-MSK-Clusters und Ihres Amazon-VPC-Netzwerks für Lambda

Um Ihre AWS Lambda Event Function mit Ihrem Amazon MSK-Cluster zu verbinden, müssen Sie Ihren Cluster und die Amazon Virtual Private Cloud (VPC), in der er sich befindet, korrekt konfigurieren. Auf dieser Seite wird beschrieben, wie Sie Ihren Cluster und Ihre VPC konfigurieren. Wenn Ihr Cluster und Ihre VPC bereits ordnungsgemäß konfiguriert sind, lesen Sie unter the section called “Konfigurieren der Ereignisquelle” nach, wie Sie die Zuordnung von Ereignisquellen konfigurieren.

Themen

- Überblick über die Anforderungen an die Netzwerkkonfiguration für Lambda- und MSK-Integrationen
- Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle
- Konfigurieren AWS PrivateLink Endpunkte für eine MSK-Ereignisquelle

Überblick über die Anforderungen an die Netzwerkkonfiguration für Lambda- und MSK-Integrationen

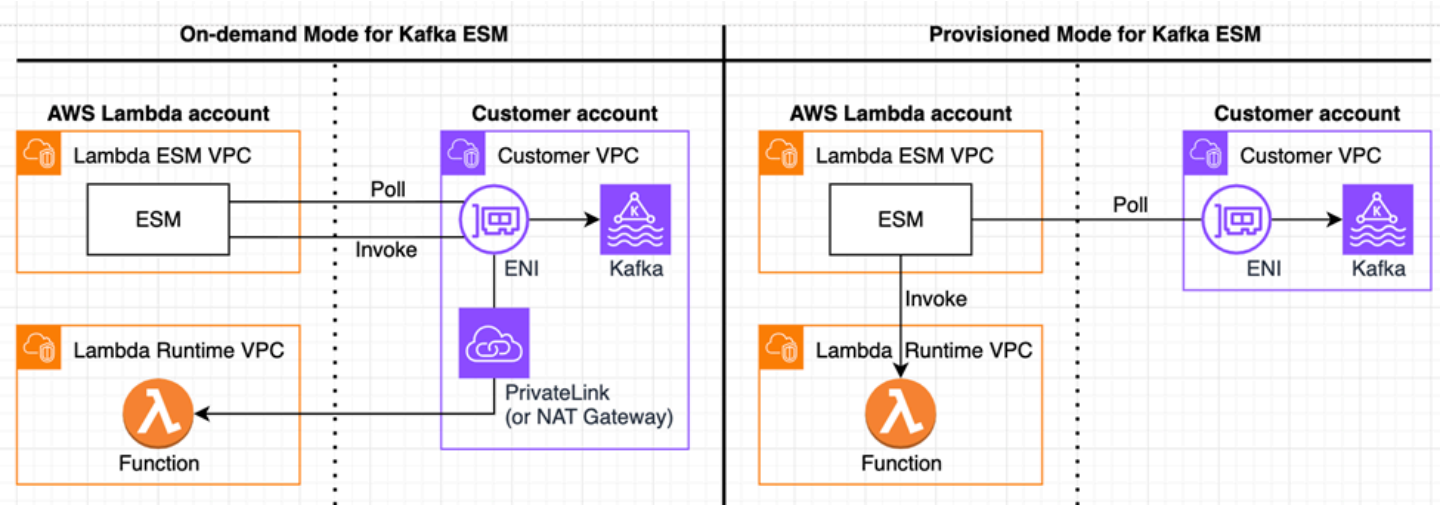
Die für eine Lambda- und MSK-Integration erforderliche Netzwerkkonfiguration hängt von der Netzwerkarchitektur Ihrer Anwendung ab. An dieser Integration sind drei Hauptressourcen beteiligt: der Amazon MSK-Cluster, die Lambda-Ereignisfunktion und die Lambda-Ereignisquellenzuordnung. Jede dieser Ressourcen befindet sich in einer anderen VPC:

- Ihr Amazon-MSK-Cluster befindet sich normalerweise in einem privaten Subnetz einer VPC, die Sie verwalten.
- Ihre Lambda-Ereignisfunktion befindet sich in einer AWS verwalteten VPC, die Lambda gehört.
- Ihre Lambda-Ereignisquellenzuordnung befindet sich in einer anderen von Lambda AWS verwalteten VPC, getrennt von der VPC, die Ihre Funktion enthält.

Die Ereignisquellenzuordnung ist die Zwischenressource zwischen dem MSK-Cluster und der Lambda-Ereignisfunktion. Die Zuordnung von Ereignisquellen hat zwei Hauptaufgaben. Zunächst

fragt sie Ihren MSK-Cluster nach neuen Nachrichten ab. Dann ruft es Ihre Lambda-Ereignisfunktion mit diesen Nachrichten auf. Da sich diese drei Ressourcen in unterschiedlichen VPCs befinden, erfordern sowohl der Abruf- als auch der Aufrufvorgang VPC-übergreifende Netzwerkaufrufe.

Die Anforderungen an die Netzwerkkonfiguration für Ihre Zuordnung von Ereignisquellen hängen davon ab, ob der Bereitstellungsmodus oder der On-Demand-Modus verwendet wird, wie im folgenden Diagramm dargestellt:



Die Art und Weise, wie die Lambda-Zuordnung von Ereignisquellen Ihren MSK-Cluster nach neuen Nachrichten abfragt, ist in beiden Modi dieselbe. Um eine Verbindung zwischen Ihrer Zuordnung von Ereignisquellen und Ihrem MSK-Cluster herzustellen, erstellt Lambda in Ihrem privaten Subnetz eine Hyperplane-ENI (oder verwendet eine vorhandene, falls verfügbar), um eine sichere Verbindung herzustellen. Wie im Diagramm dargestellt, verwendet dieses Hyperplane-ENI die Subnetz- und Sicherheitsgruppenkonfiguration Ihres MSK-Clusters, nicht Ihre Lambda Event Function.

Nachdem Sie die Nachricht vom Cluster abgerufen haben, ist die Art und Weise, wie Lambda Ihre Funktion aufruft, in jedem Modus unterschiedlich:

- Im Bereitstellungsmodus verarbeitet Lambda automatisch die Verbindung zwischen der VPC für die Zuordnung von Ereignisquellen und der Funktions-VPC. Sie benötigen also keine zusätzlichen Netzwerkkomponenten, um Ihre Funktion erfolgreich aufzurufen.
- Im On-Demand-Modus ruft Ihre Lambda-Ereignisquellenzuordnung Ihre Funktion auf, indem sie einen Pfad durch Ihre vom Kunden verwaltete VPC verwendet. Aus diesem Grund müssen Sie entweder ein NAT-Gateway im öffentlichen Subnetz Ihrer VPC oder AWS PrivateLink-Endpunkte im privaten Subnetz der VPC konfigurieren, die Zugriff auf Lambda, AWS -Security-Token-Service (STS) und optional AWS Secrets Manager bieten. Die korrekte Konfiguration einer dieser Optionen

ermöglicht eine Verbindung zwischen Ihrer VPC und der Lambda-managed Runtime-VPC, die zum Aufrufen Ihrer Funktion erforderlich ist.

Ein NAT-Gateway ermöglicht Ressourcen in Ihrem privaten Subnetz den Zugriff auf das öffentliche Internet. Wenn Sie diese Konfiguration verwenden, durchquert Ihr Datenverkehr das Internet, bevor die Lambda-Ereignisfunktion aufgerufen wird. AWS PrivateLink Endpunkte ermöglichen es privaten Subnetzen, sich sicher mit AWS Diensten oder anderen privaten VPC-Ressourcen zu verbinden, ohne das öffentliche Internet zu durchqueren. Weitere Informationen zur Konfiguration dieser Ressourcen finden Sie unter the section called “Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle” oder the section called “Konfigurieren AWS PrivateLink Endpunkte für eine MSK-Ereignisquelle”.

Bisher gingen wir davon aus, dass sich Ihr MSK-Cluster in einem privaten Subnetz innerhalb Ihrer VPC befindet, was der häufigste Fall ist. Selbst wenn sich Ihr MSK-Cluster in einem öffentlichen Subnetz innerhalb Ihrer VPC befindet, müssen Sie AWS PrivateLink -Endpunkte konfigurieren, um eine sichere Verbindung zu gewährleisten. In der folgenden Tabelle sind die Anforderungen an die Netzwerkkonfiguration zusammengefasst, je nachdem, wie Sie Ihren MSK-Cluster und die Lambda-Zuordnung von Ereignisquellen konfigurieren:

Standort des MSK-Clusters (in der vom Kunden verwalteten VPC)	Skalierungsmodus der Lambda-Zuordnung von Ereignisquellen	Erforderliche Netzwerkkonfiguration
Privates Subnetz	On-demand Modus	NAT-Gateway (im öffentlichen Subnetz Ihrer VPC) oder AWS PrivateLink Endpoints (im privaten Subnetz Ihrer VPC), um den Zugriff auf Lambda und optional auf Secrets Manager zu ermöglichen. AWS STS
Öffentliches Subnetz	On-demand Modus	AWS PrivateLink Endpunkte (im öffentlichen Subnetz Ihrer VPC), um den Zugriff auf Lambda und optional auf

Standort des MSK-Clusters (in der vom Kunden verwalteten VPC)	Skalierungsmodus der Lambda-Zuordnung von Ereignisquellen	Erforderliche Netzwerkkonfiguration
		Secrets AWS STS Manager zu ermöglichen.
Privates Subnetz	Modus bereitgestellter Kapazität	Keine
Öffentliches Subnetz	Modus bereitgestellter Kapazität	Keine

Darüber hinaus müssen die mit Ihrem MSK-Cluster verknüpften Sicherheitsgruppen den Datenverkehr über die richtigen Ports zulassen. Stellen Sie sicher, dass Sie die folgenden Regeln für Sicherheitsgruppen konfiguriert haben:

- Regeln für eingehenden Datenverkehr – Erlauben Sie den gesamten Datenverkehr auf dem standardmäßigen Broker-Port. Der Port, den MSK verwendet, hängt von der Art der Authentifizierung auf dem Cluster ab: 9098 für die IAM-Authentifizierung, 9096 für und für TLS. SASL/SCRAM 9094 Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr — Lassen Sie den gesamten Datenverkehr über den Port 443 für externe Ziele zu, wenn Ihre Funktion mit anderen Diensten kommunizieren muss. AWS Alternativ können Sie eine selbstreferenzierende Sicherheitsgruppenregel verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen Diensten kommunizieren müssen. AWS
- Regeln für eingehenden Amazon VPC-Endpunkt — Wenn Sie einen Amazon VPC-Endpunkt verwenden, muss die dem Endpunkt zugeordnete Sicherheitsgruppe eingehenden Datenverkehr über den Port 443 von der Sicherheitsgruppe des Clusters zulassen.

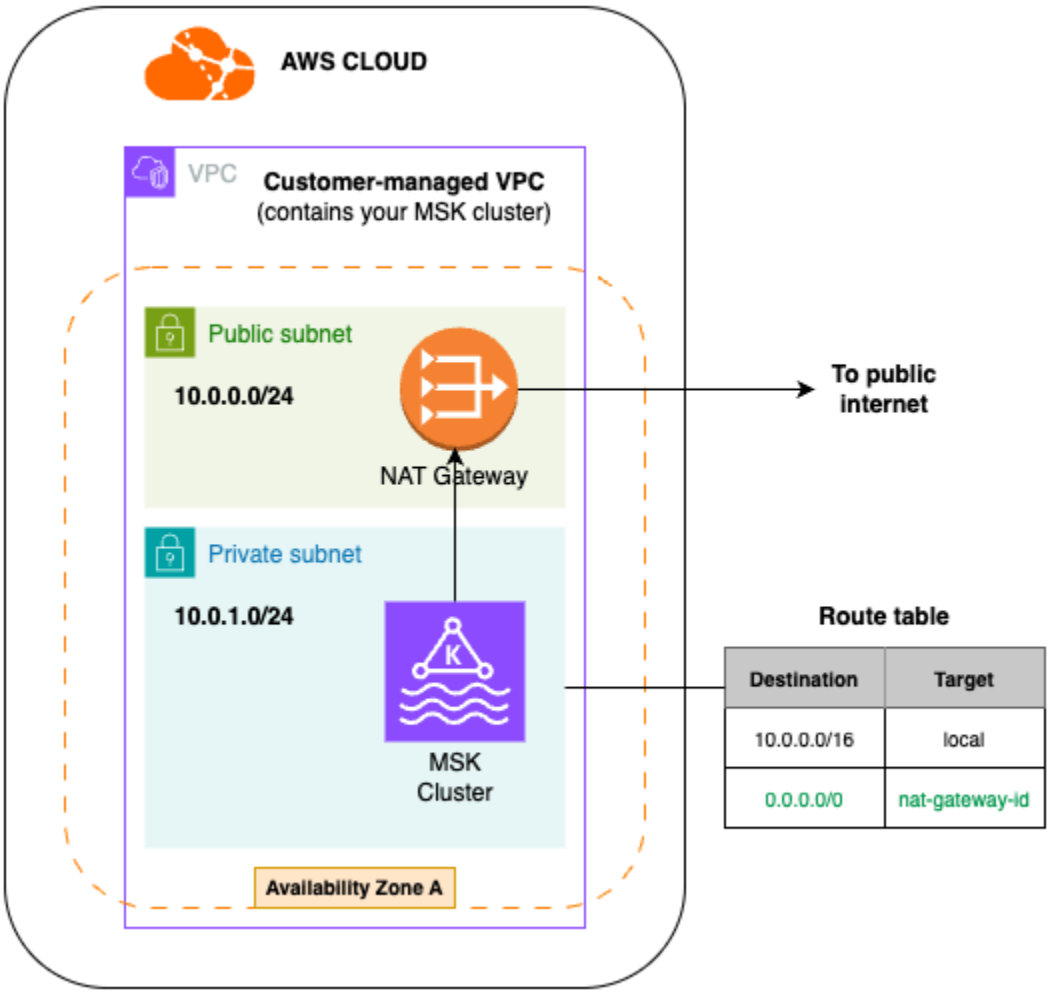
Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle

Sie können ein NAT-Gateway so konfigurieren, dass Ihre Ereignisquellenzuordnung Nachrichten von Ihrem Cluster abfragen und die Event-Funktion mithilfe eines Pfads durch Ihre VPC aufrufen kann. Dies ist nur erforderlich, wenn Ihre Zuordnung von Ereignisquellen den On-Demand-Modus

verwendet und sich Ihr Cluster in einem privaten Subnetz Ihrer VPC befindet. Wenn sich Ihr Cluster in einem öffentlichen Subnetz Ihrer VPC befindet oder Ihre Ereignisquellenzuordnung den Bereitstellungsmodus verwendet, müssen Sie kein NAT-Gateway konfigurieren.

Ein NAT-Gateway ermöglicht Ressourcen in einem privaten Subnetz den Zugriff auf das öffentliche Internet. Wenn Sie eine private Verbindung zu Lambda benötigen, finden Sie stattdessen weitere Informationen unter the section called “Konfigurieren AWS PrivateLink Endpunkte für eine MSK-Ereignisquelle”.

Nachdem Sie Ihr NAT-Gateway konfiguriert haben, müssen Sie die entsprechenden Routing-Tabellen konfigurieren. Dadurch kann der Datenverkehr aus Ihrem privaten Subnetz über das NAT-Gateway in das öffentliche Internet geleitet werden.



Die folgenden Schritte führen Sie durch die Konfiguration eines NAT-Gateways über die Konsole. Wiederholen Sie diese Schritte bei Bedarf für jede Availability Zone (AZ).

So konfigurieren Sie ein NAT-Gateway und das richtige Routing (Konsole)

1. Folgen Sie den Schritten unter Erstellen eines NAT-Gateways und beachten Sie dabei Folgendes:

   - NAT-Gateways sollten sich immer in einem öffentlichen Subnetz befinden. Erstellen Sie NAT-Gateways mit öffentlicher Konnektivität.
   - Wenn Ihr MSK-Cluster über mehrere AZs repliziert wird, erstellen Sie ein NAT-Gateway pro AZ. In jeder AZ sollte Ihre VPC beispielsweise ein privates Subnetz mit Ihrem Cluster und ein öffentliches Subnetz mit Ihrem NAT-Gateway haben. Bei einem Setup mit drei AZs verfügen Sie über drei private Subnetze, drei öffentliche Subnetze und drei NAT-Gateways.
2. Nachdem Sie Ihr NAT-Gateway erstellt haben, öffnen Sie die Amazon VPC-Konsole und wählen Sie im linken Menü Routing-Tabellen aus.
3. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
4. Ordnen Sie diese Routing-Tabelle der VPC zu, die Ihren MSK-Cluster enthält. Geben Sie optional einen Namen für Ihre Routing-Tabelle ein.
5. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
6. Wählen Sie die Routing-Tabelle aus, die Sie gerade erstellt haben.
7. Wählen Sie auf der Registerkarte Subnetzzuordnungen die Option Subnetzzuordnungen bearbeiten aus.

   - Ordnen Sie diese Routing-Tabelle dem privaten Subnetz zu, das Ihren MSK-Cluster enthält.
8. Wählen Sie Routen bearbeiten aus.
9. Wählen Sie Route hinzufügen aus:

   1. Geben Sie für Ziel die Zeichenfolge `0.0.0.0/0` an.
   2. Wählen Sie für Ziel die Option NAT-Gateway aus.
   3. Wählen Sie im Suchfeld das in Schritt 1 erstellte NAT-Gateway aus. Dies sollte das NAT-Gateway in derselben AZ sein wie das private Subnetz, das Ihren MSK-Cluster enthält (das private Subnetz, das Sie dieser Routing-Tabelle in Schritt 6 zugeordnet haben).
10. Wählen Sie Änderungen speichern aus.

Konfigurieren AWS PrivateLink Endpunkte für eine MSK-Ereignisquelle

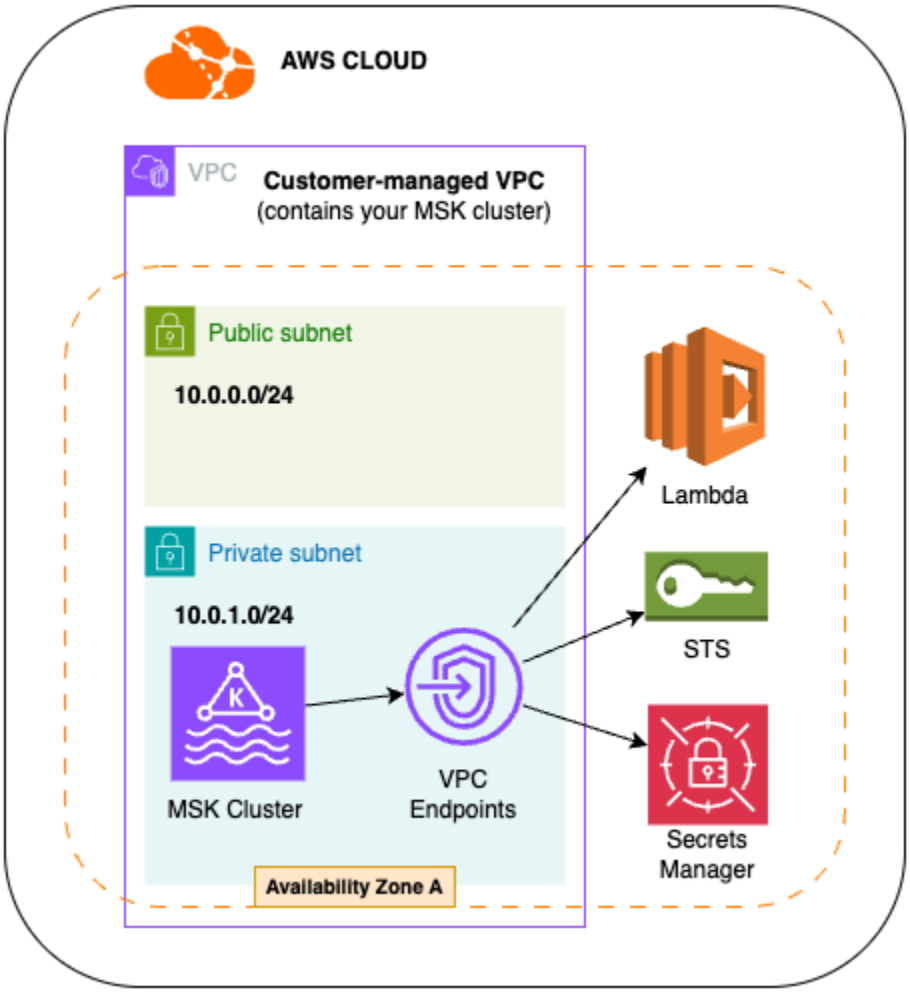
Sie können AWS PrivateLink Endpoints so konfigurieren, dass sie Nachrichten von Ihrem Cluster abfragen, und die Funktion aufrufen, indem Sie einen Pfad durch Ihre VPC verwenden. Diese Endpunkte sollten Ihrem MSK-Cluster den Zugriff auf Folgendes ermöglichen:

- Den Lambda-Service
- Den AWS -Security-Token-Service (STS)
- Optional den AWS Secrets Manager-Service. Dies ist erforderlich, wenn das für die Cluster-Authentifizierung erforderliche Secret im Secrets Manager gespeichert ist.

Die Konfiguration von PrivateLink Endpunkten ist nur erforderlich, wenn Ihre Ereignisquellenzuordnung den On-Demand-Modus verwendet. Wenn Ihre Zuordnung von Ereignisquellen den Bereitstellungsmodus verwendet, richtet Lambda die erforderlichen Verbindungen für Sie ein.

PrivateLink Endpunkte ermöglichen einen sicheren, privaten Zugriff auf AWS Dienste über. AWS PrivateLink Alternativ können Sie ein NAT-Gateway konfigurieren, um Ihrem MSK-Cluster Zugriff auf das öffentliche Internet zu ermöglichen (siehe the section called “Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle”).

Nachdem Sie Ihre VPC-Endpunkte konfiguriert haben, sollte Ihr MSK-Cluster direkten und privaten Zugriff auf Lambda, STS und optional Secrets Manager haben.



Die folgenden Schritte führen Sie durch die Konfiguration eines PrivateLink Endpunkts mithilfe der Konsole. Wiederholen Sie diese Schritte nach Bedarf für jeden Endpunkt (Lambda, STS, Secrets Manager).

So konfigurieren Sie PrivateLink VPC-Endpunkte (Konsole)

1. Öffnen Sie die Amazon VPC-Konsole und wählen Sie im linken Menü Endpunkte aus.
2. Wählen Sie Endpunkt erstellen aus.
3. Geben Sie optional einen Namen für Ihren Endpunkt ein.
4. Wählen Sie für Typ die Option AWS -Services aus.
5. Beginnen Sie unter Services mit der Eingabe des Namens des Service. Um beispielsweise einen Endpunkt für die Verbindung mit Lambda zu erstellen, geben Sie `lambda` in das Suchfeld ein.

6. In den Ergebnissen sollte der Service-Endpunkt in der aktuellen Region angezeigt werden. In der Region USA Ost (Nord-Virginia) sollte beispielsweise `com.amazonaws.us-east-2.lambda` angezeigt werden. Wählen Sie diesen Service aus.
7. Wählen Sie unter Netzwerkeinstellungen die VPC aus, die Ihren MSK-Cluster enthält.
8. Wählen Sie unter Subnetze die AZs aus, in denen sich Ihr MSK-Cluster befindet.
   - Wählen Sie für jede AZ unter Subnetz-ID das private Subnetz aus, das Ihren MSK-Cluster enthält.
9. Wählen Sie unter Sicherheitsgruppen die Ihrem MSK-Cluster zugeordneten Sicherheitsgruppen aus.
10. Wählen Sie Endpunkt erstellen aus.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Für Ihren Secrets Manager Manager-Endpunkt können Sie beispielsweise dessen Richtlinie so ändern, dass nur die Ausführungsrolle Ihrer Funktion auf das Geheimnis zugreifen kann.

Example VPC-Endpunktrichtlinie – Secrets-Manager-Endpunkt

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

Für die AWS STS und Lambda-Endpunkte können Sie den aufrufenden Principal auf den Lambda-Serviceprinzipal beschränken. Stellen Sie jedoch sicher, dass Sie in diesen Richtlinien `"Resource": "*"` verwenden.

Example VPC-Endpunktrichtlinie — AWS STS Endpunkt

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Konfiguration von Lambda-Berechtigungen für Amazon MSK-Ereignisquellenzuordnungen

Für den Zugriff auf den Amazon MSK-Cluster benötigen Ihre Ereignisfunktion und die Zuordnung der Ereignisquelle Berechtigungen zur Ausführung verschiedener Amazon MSK-API-Aktionen. Fügen Sie diese Berechtigungen zur Ausführungsrolle der Funktion hinzu. Wenn Ihre Benutzer Zugriff benötigen, fügen Sie die erforderlichen Berechtigungen zur Identitätsrichtlinie für Benutzer oder Rolle hinzu.

Die AWSLambdaMSKExecutionRoleverwaltete Richtlinie enthält die erforderlichen Mindestberechtigungen für Amazon MSK Lambda-Ereignisquellenzuordnungen. Um den Genehmigungsprozess zu vereinfachen, können Sie:

- Fügen Sie die AWSLambdaMSKExecutionRoleverwaltete Richtlinie Ihrer Ausführungsrolle hinzu.
- Lassen Sie die Lambda-Konsole die Berechtigungen für Sie generieren. Wenn Sie eine Amazon MSK-Ereignisquellenzuordnung in der Konsole erstellen, bewertet Lambda Ihre Ausführungsrolle und warnt Sie, wenn Berechtigungen fehlen. Wählen Sie Berechtigungen generieren, um Ihre Ausführungsrolle automatisch zu aktualisieren. Dies funktioniert nicht, wenn Sie Ihre Richtlinien für Ausführungsrollen manuell erstellt oder geändert haben oder wenn die Richtlinien mehreren Rollen zugeordnet sind. Beachten Sie, dass in Ihrer Ausführungsrolle möglicherweise weiterhin zusätzliche Berechtigungen erforderlich sind, wenn Sie erweiterte Funktionen wie On-Failure Destination oder AWS Glue Schema Registry verwenden.

Themen

- Erforderliche Berechtigungen
- Optionale Berechtigungen

Erforderliche Berechtigungen

Ihre Ausführungsrolle für Lambda Event Function muss über die folgenden erforderlichen Berechtigungen für Amazon MSK-Ereignisquellenzuordnungen verfügen. Diese Berechtigungen sind in der verwalteten Richtlinie enthalten. AWSLambdaMSKExecutionRole

CloudWatch Protokolliert Berechtigungen

Die folgenden Berechtigungen ermöglichen es Lambda, Protokolle in Amazon CloudWatch Logs zu erstellen und zu speichern.

- Protokolle: CreateLogGroup
- Protokolle: CreateLogStream
- Protokolle: PutLogEvents

MSK-Cluster-Berechtigungen

Die folgenden Berechtigungen ermöglichen Lambda, in Ihrem Namen auf Ihren Amazon MSK-Cluster zuzugreifen:

- Kafka: DescribeCluster
- Kafka: V2 DescribeCluster
- Kafka: GetBootstrapBrokers

Wir empfehlen die Verwendung von kafka: DescribeCluster V2 anstelle von kafka:. DescribeCluster Die v2-Berechtigung funktioniert sowohl mit bereitgestellten als auch mit serverlosen Amazon MSK-Clustern. In Ihrer Richtlinie benötigen Sie nur eine dieser Berechtigungen.

VPC-Berechtigungen

Die folgenden Berechtigungen ermöglichen es Lambda, Netzwerkschnittstellen zu erstellen und zu verwalten, wenn eine Verbindung zu Ihrem Amazon MSK-Cluster hergestellt wird:

- ec2: CreateNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeVpcs
- ec2: DeleteNetworkInterface
- ec2: DescribeSubnets
- ec2: DescribeSecurityGroups

Optionale Berechtigungen

Ihre Lambda-Ereignisfunktion benötigt möglicherweise auch Berechtigungen für:

- Greifen Sie auf kontenübergreifende Amazon MSK-Cluster zu. Für kontenübergreifende Zuordnungen von Ereignisquellen benötigen Sie kafka: in der Ausführungsrolle. DescribeVpcConnection Ein IAM-Prinzipal, der eine kontenübergreifende Zuordnung von Ereignisquellen erstellt, benötigt kafka:. ListVpcConnections
- Greifen Sie auf Ihr SCRAM-Geheimnis zu, wenn Sie die Authentifizierung verwenden. SASL/SCRAM Damit kann Ihre Funktion einen Benutzernamen und ein Passwort verwenden, um eine Verbindung zu Kafka herzustellen.
- Beschreiben Sie Ihr Secrets Manager-Geheimnis, falls Sie die SASL/SCRAM mTLS-Authentifizierung verwenden. Mit dieser Berechtigung kann Ihre Funktion die Anmeldeinformationen oder Zertifikate abrufen, die für sichere Verbindungen benötigt werden.
- Greifen Sie auf Ihren vom AWS KMS Kunden verwalteten Schlüssel zu, wenn Ihr AWS Secrets Manager Geheimnis mit einem vom AWS KMS Kunden verwalteten Schlüssel verschlüsselt ist.

- Greifen Sie auf Ihre Schemaregistrierungs-Secrets zu, wenn Sie eine Schemaregistrierung mit Authentifizierung verwenden:
  - Für AWS Glue Schema Registry: Ihre Funktionsanforderungen `glue:GetRegistry` und `glue:GetSchemaVersion` Berechtigungen. Dadurch kann Ihre Funktion die in AWS Glue gespeicherten Nachrichtenformatregeln nachschlagen und verwenden.
  - Für Confluent Schema Registry mit `BASIC_AUTH` oder`CLIENT_CERTIFICATE_TLS_AUTH`: Ihre Funktion benötigt die Berechtigung `secretsmanager:GetSecretValue` für das Secret, das die Authentifizierungsdaten enthält. Dadurch kann Ihre Funktion die Oder-Zertifikate abrufen, die username/password für den Zugriff auf die Confluent Schema Registry erforderlich sind.
  - Für private CA-Zertifikate: Ihre Funktion benötigt die GetSecretValue Berechtigung secretsmanager: für das Secret, das das Zertifikat enthält. Mit dieser Berechtigung kann Ihre Funktion die Identität von Schemaregistern überprüfen, die benutzerdefinierte Zertifikate verwenden.
- Greifen Sie auf Kafka-Cluster-Nutzergruppen zu und fragen Sie Nachrichten aus dem Thema ab, wenn Sie die IAM-Authentifizierung für die Zuordnung der Ereignisquellen verwenden.

Diese entsprechen den folgenden erforderlichen Berechtigungen:

- kafka: ListScramSecrets — Ermöglicht die Auflistung von SCRAM-Geheimnissen für die Kafka-Authentifizierung
- secretsmanager: GetSecretValue - Ermöglicht das Abrufen von Geheimnissen aus Secrets Manager
- kms:Decrypt — Ermöglicht die Entschlüsselung verschlüsselter Daten mit AWS KMS
- glue: GetRegistry - Ermöglicht den Zugriff auf die Schemaregistrierung AWS Glue
- glue: GetSchemaVersion - Ermöglicht das Abrufen bestimmter Schemaversionen aus der AWS Glue Schemaregistry
- kafka-cluster:Connect — Erteilt die Erlaubnis, eine Verbindung zum Cluster herzustellen und sich dort zu authentifizieren
- kafka-cluster: AlterGroup - Erteilt die Erlaubnis, Gruppen in einem Cluster beizutreten, was der READ GROUP-ACL von Apache Kafka entspricht
- kafka-cluster: DescribeGroup - Erteilt die Erlaubnis, Gruppen in einem Cluster zu beschreiben, was der DESCRIBE GROUP-ACL von Apache Kafka entspricht
- kafka-cluster: DescribeTopic - Erteilt die Erlaubnis, Themen in einem Cluster zu beschreiben, was der DESCRIBE TOPIC-ACL von Apache Kafka entspricht

- kafka-cluster: ReadData - Erteilt die Berechtigung, Daten aus Themen in einem Cluster zu lesen, was der READ TOPIC-ACL von Apache Kafka entspricht

Wenn Sie außerdem Aufzeichnungen fehlgeschlagener Aufrufe an ein Ziel für Fehlerfälle senden möchten, benötigen Sie je nach Zieltyp die folgenden Berechtigungen:

- Für Amazon SQS SQS-Ziele: sqs: SendMessage — Ermöglicht das Senden von Nachrichten an eine Amazon SQS SQS-Warteschlange
- Für Amazon-SNS-Ziele: sns:Publish – erlaubt die Veröffentlichung von Nachrichten zu einem Amazon-SNS-Thema
- Für Amazon S3 S3-Bucket-Ziele: s3: PutObject und s3: ListBucket — Ermöglicht das Schreiben und Auflisten von Objekten in einem Amazon S3 S3-Bucket

Informationen zur Behebung von Authentifizierungs- und Autorisierungsfehlern finden Sie unter the section called “Fehlerbehebung”.

Konfigurieren von Amazon MSK-Ereignisquellen für Lambda

Um einen Amazon MSK-Cluster als Ereignisquelle für Ihre Lambda-Ereignisfunktion zu verwenden, erstellen Sie eine Ereignisquellenzuordnung, die die beiden Ressourcen verbindet. Auf dieser Seite wird beschrieben, wie Sie eine Zuordnung von Ereignisquellen für Amazon MSK erstellen.

Auf dieser Seite wird davon ausgegangen, dass Sie Ihren MSK-Cluster und die Amazon Virtual Private Cloud (VPC), in der er sich befindet, bereits ordnungsgemäß konfiguriert haben. Wenn Sie Ihren Cluster oder Ihre VPC einrichten müssen, finden Sie entsprechende Informationen unter the section called “Einrichtung von Cluster und Netzwerk”. Informationen zum Konfigurieren des Wiederholungsverhaltens für die Fehlerbehandlung finden Sie unter. the section called “Versuchen Sie es erneut mit den Konfigurationen”

Themen

- Verwenden eines Amazon MSK-Clusters als Ereignisquelle
- Konfiguration von Amazon MSK-Cluster-Authentifizierungsmethoden in Lambda
- Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine Amazon-MSK-Ereignisquelle
- Erstellen von kontenübergreifenden Zuordnungen von Ereignisquellen in Lambda
- Alle Amazon-MSK-Konfigurationsparameter von Ereignisquellen in Lambda

Verwenden eines Amazon MSK-Clusters als Ereignisquelle

Wenn Sie Ihren Apache Kafka- oder Amazon MSK-Cluster als Auslöser für Ihre Lambda Event Function hinzufügen, wird der Cluster als Ereignisquelle verwendet.

Lambda liest Ereignisdaten aus den Kafka-Themen, die Sie wie `Topics` in einer CreateEventSourceMappingAnfrage angeben, basierend auf der von Ihnen angegebenen Startposition. Nach erfolgreicher Verarbeitung wird Ihr Kafka-Thema Ihrem Kafka-Cluster zugeordnet.

Lambda liest Nachrichten sequentiell für jede Kafka-Themenpartition. Eine einzelne Lambda-Nutzlast kann Nachrichten von mehreren Partitionen enthalten. Wenn mehr Datensätze verfügbar sind, setzt Lambda die Verarbeitung von Datensätzen stapelweise fort, basierend auf dem BatchSize Wert, den Sie in einer CreateEventSourceMappingAnfrage angeben, bis Ihre Funktion das Thema eingeholt hat.

Nachdem Lambda jeden Batch verarbeitet hat, werden die Offsets der Nachrichten in diesem Batch festgeschrieben. Wenn Ihre Funktion einen Fehler für eine der Nachrichten in einem Batch zurückgibt, wiederholt Lambda den gesamten Nachrichtenbatch, bis die Verarbeitung erfolgreich ist oder die Nachrichten ablaufen. Sie können Datensätze, bei denen alle Wiederholungsversuche fehlschlagen, zur späteren Verarbeitung an ein Ausfallziel senden.

> **Note**
>
> Während Lambda-Ereignisfunktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB und Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten.

Konfiguration von Amazon MSK-Cluster-Authentifizierungsmethoden in Lambda

Lambda benötigt eine Berechtigung, um auf Ihren Amazon-MSK-Cluster zuzugreifen, Datensätze abzurufen und andere Aufgaben auszuführen. Amazon MSK unterstützt mehrere Methoden zur Authentifizierung bei Ihrem MSK-Cluster.

Cluster-Authentifizierungsmethoden

- Nicht authentifizierter Zugriff
- SASL/SCRAM authentication
- Gegenseitige TLS-Authentifizierung

- IAM-Authentifizierung
- So wählt Lambda einen Bootstrap-Broker

Nicht authentifizierter Zugriff

Wenn keine Clients über das Internet auf den Cluster zugreifen, können Sie einen nicht authentifizierten Zugriff verwenden.

SASL/SCRAM authentication

Lambda unterstützt die Authentifizierung Simple Authentication und Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) mit der SHA-512 Hash-Funktion und der Transport Layer Security (TLS) -Verschlüsselung. Damit Lambda eine Verbindung zum Cluster herstellen kann, speichern Sie die Authentifizierungsanmeldeinformationen (Benutzername und Passwort) in einem Secrets-Manager-Geheimnis und verweisen Sie bei der Konfiguration Ihrer Zuordnung von Ereignisquellen auf dieses Geheimnis.

Weitere Informationen zur Verwendung von Secrets Manager finden Sie unter Authentifizierung von Sign-in Anmeldeinformationen mit Secrets Manager im Amazon Managed Streaming for Apache Kafka Developer Guide.

> **Note**
>
> Amazon MSK unterstützt keine SASL/PLAIN Authentifizierung.

Gegenseitige TLS-Authentifizierung

Gegenseitige TLS (mTLS) bietet eine bidirektionale Authentifizierung zwischen Client und Server. Der Client sendet ein Zertifikat an den Server, damit dieser den Client überprüfen kann. Der Server sendet ebenfalls ein Zertifikat an den Client, damit dieser den Server überprüfen kann.

Bei Amazon-MSK-Integrationen mit Lambda fungiert Ihr MSK-Cluster als Server und Lambda als Client.

- Damit Lambda Ihren MSK-Cluster überprüfen kann, konfigurieren Sie ein Client-Zertifikat als Geheimnis in Secrets Manager und verweisen Sie in Ihrer Konfiguration der Zuordnung von Ereignisquellen auf dieses Zertifikat. Das Clientzertifikat muss von einer Zertifizierungsstelle (CA) im Trust Store des Servers signiert sein.

- Der MSK-Cluster sendet auch ein Serverzertifikat an Lambda. Das Serverzertifikat muss von einer Zertifizierungsstelle (CA) im AWS Trust Store signiert sein.

Amazon MSK unterstützt keine selbstsignierten Serverzertifikate. Alle Broker in Amazon MSK verwenden öffentliche Zertifikate, die von Amazon-Trust-Services-Endpunkt-Zertifizierungsstellen signiert sind, denen Lambda standardmäßig vertraut.

Konfigurieren des mTLS-Secrets

Das Secret CLIENT_CERTIFICATE_TLS_AUTH erfordert ein Zertifikatfeld und ein Feld für einen privaten Schlüssel. Für einen verschlüsselten privaten Schlüssel erfordert das Secret ein Passwort für den privaten Schlüssel. Sowohl das Zertifikat als auch der private Schlüssel müssen im PEM-Format vorliegen.

> **Note**
>
> Lambda unterstützt die PBES1-Algorithmen (aber nicht PBES2) zur Verschlüsselung von privaten Schlüsseln.

Das Zertifikatfeld muss eine Liste von Zertifikaten enthalten, beginnend mit dem Client-Zertifikat, gefolgt von etwaigen Zwischenzertifikaten und endend mit dem Root-Zertifikat. Jedes Zertifikat muss in einer neuen Zeile mit der folgenden Struktur beginnen:

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager unterstützt Secrets von bis zu 65 536 Bytes, was genügend Platz für lange Zertifikatsketten bietet.

Der private Schlüssel muss im Format PKCS #8 mit folgender Struktur vorliegen:

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

Verwenden Sie für einen verschlüsselten privaten Schlüssel die folgende Struktur:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

Im folgenden Beispiel sehen Sie den Inhalt eines Secrets für mTLS-Authentifizierung mit einem verschlüsselten privaten Schlüssel. Fügen Sie für einen verschlüsselten privaten Schlüssel das Passwort für den privaten Schlüssel in das Secret ein.

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Weitere Informationen zu mTLS für Amazon MSK und Anweisungen zum Generieren eines Clientzertifikats finden Sie unter Gegenseitige TLS-Authentifizierung für Amazon MSK im Entwicklerhandbuch für Amazon Managed Streaming for Apache Kafka.

IAM-Authentifizierung

Sie können AWS Identity and Access Management (IAM) verwenden, um die Identität von Clients zu authentifizieren, die eine Verbindung zum MSK-Cluster herstellen. Bei der IAM-Authentifizierung stützt sich Lambda auf die Berechtigungen in der Ausführungsrolle Ihrer Event-Funktion, um eine Verbindung zum Cluster herzustellen, Datensätze abzurufen und andere erforderliche Aktionen auszuführen. Ein Beispiel für eine Richtlinie, welche die erforderlichen Berechtigungen enthält, finden

Sie unter Erstellen von Autorisierungsrichtlinien für die IAM-Rolle im Entwicklerhandbuch für Amazon Managed Streaming for Apache Kafka.

Wenn die IAM-Authentifizierung für Ihren MSK-Cluster aktiv ist und Sie kein Geheimnis angeben, verwendet Lambda automatisch standardmäßig die IAM-Authentifizierung.

Weitere Informationen zur IAM-Authentifizierung in Amazon MSK finden Sie unter IAM-Zugriffskontrolle.

So wählt Lambda einen Bootstrap-Broker

Lambda wählt einen Bootstrap-Broker basierend auf den auf Ihrem Cluster verfügbaren Authentifizierungsmethoden aus und ob Sie ein Geheimnis für die Authentifizierung angeben.. Wenn Sie ein Geheimnis für mTLS oder angeben SASL/SCRAM, wählt Lambda automatisch diese Authentifizierungsmethode. Wenn Sie kein Geheimnis angeben, wählt Lambda die stärkste Authentifizierungsmethode aus, die in Ihrem Cluster aktiv ist. Im Folgenden ist die Reihenfolge der Priorität aufgeführt, in der Lambda einen Broker auswählt, von der stärksten zur schwächsten Authentifizierung:

- mTLS (Geheimnis für mTLS bereitgestellt)
- SASL/SCRAM (Geheimnis vorgesehen) SASL/SCRAM
- SASL IAM (kein Geheimnis angegeben und IAM-Authentifizierung aktiv)
- Nicht authentifiziertes TLS (kein Geheimnis bereitgestellt und IAM-Authentifizierung nicht aktiv)
- Klartext (kein Geheimnis angegeben und sowohl IAM-Authentifizierung als auch nicht authentifiziertes TLS sind nicht aktiv)

> **Note**
>
> Wenn Lambda keine Verbindung zum sichersten Brokertyp herstellen kann, versucht Lambda nicht, eine Verbindung zu einem anderen (schwächeren) Brokertyp herzustellen. Wenn Sie möchten, dass Lambda einen schwächeren Brokertyp wählt, deaktivieren Sie alle stärkeren Authentifizierungsmethoden in Ihrem Cluster.

Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine Amazon-MSK-Ereignisquelle

Zum Erstellen einer Zuordnung von Ereignisquellen können Sie die Lambda-Konsole, die AWS Command Line Interface (CLI) oder ein AWS SDK verwenden.

> **Note**
>
> Wenn Sie die Zuordnung von Ereignisquellen erstellen, erstellt Lambda eine Hyperplane-ENI im privaten Subnetz, das Ihren MSK-Cluster enthält, sodass Lambda eine sichere Verbindung herstellen kann. Diese Hyperplane-ENI ermöglicht es, die Subnetz- und Sicherheitsgruppenkonfiguration Ihres MSK-Clusters zu verwenden, nicht Ihre Lambda Event Function.

Mit den folgenden Konsolenschritten wird ein Amazon MSK-Cluster als Auslöser für Ihre Lambda Event Function hinzugefügt. Im Hintergrund wird dadurch eine Zuordnung von Ereignisquellen erstellt.

So fügen Sie Ihrer Lambda Event Function (Konsole) einen Amazon MSK-Trigger hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen der Lambda-Ereignisfunktion, zu der Sie einen Amazon MSK-Trigger hinzufügen möchten.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie unter Auslöser-Konfiguration die Option MSK aus.
5. Führen Sie die folgenden Schritte aus, um Ihre Kafka-Cluster-Details anzugeben:
    1. Wählen Sie für MSK-Cluster Ihren Cluster aus.
    2. Geben Sie für Themenname den Namen des Kafka-Themas ein, von dem Nachrichten verbraucht werden sollen.
    3. Geben Sie für Verbrauchergruppen-ID ggf. die ID einer Kafka-Verbrauchergruppe ein, der Sie beitreten möchten. Weitere Informationen finden Sie unter the section called “Verbrauchergruppen-ID”.
6. Nehmen Sie für die Cluster-Authentifizierung die erforderlichen Konfigurationen vor. Weitere Informationen über Cluster-Authentifizierung finden Sie unter the section called “Cluster-Authentifizierung”.
    - Aktivieren Sie Authentifizierung verwenden, wenn Lambda beim Verbindungsaufbau die Authentifizierung bei Ihrem MSK-Cluster durchführen soll. Die Authentifizierung wird empfohlen.

- Wenn Sie Authentifizierung verwenden, wählen Sie für Authentifizierungsmethode die zu verwendende Authentifizierungsmethode aus.
- Wenn Sie Authentifizierung verwenden, wählen Sie für Secrets-Manager-Schlüssel den Secrets-Manager-Schlüssel aus, der die für den Zugriff auf Ihren Cluster erforderlichen Authentifizierungsanmeldeinformationen enthält.

7. Nehmen Sie unter Konfiguration des Ereignis-Pollers die erforderlichen Konfigurationen vor.

   - Wählen Sie Auslöser aktivieren, um den Auslöser unmittelbar nach der Erstellung zu aktivieren.
   - Wählen Sie aus, ob Sie den Bereitstellungsmodus für Ihre Zuordnung von Ereignisquellen konfigurieren möchten. Weitere Informationen finden Sie unter the section called “Skalierung des Ereignis-Pollers”.
     - Wenn Sie den Bereitstellungsmodus konfigurieren, geben Sie einen Wert für Minimum an Event Pollers, einen Wert für Maximum Event Pollers und einen optionalen Wert für PollerGroupName die Gruppierung mehrerer ESMs innerhalb derselben Eventquellen-VPC ein.
   - Wählen Sie unter Startposition aus, wie Lambda mit dem Lesen aus Ihrem Stream beginnen soll. Weitere Informationen finden Sie unter the section called “Abfrage und Stream-Positionen”.

8. Nehmen Sie unter Batching die erforderlichen Konfigurationen vor. Weitere Informationen zu Batching finden Sie unter the section called “Batching-Verhalten”.

   1. Geben Sie für Batchgröße die maximale Anzahl von Nachrichten ein, die in einem einzelnen Batch empfangen werden sollen.
   2. Geben Sie für Batch-Fenster die maximale Zeit in Sekunden ein, die Lambda mit dem Sammeln von Datensätzen verbringt, bevor die Funktion aufgerufen wird.

9. Nehmen Sie unter Filterung die erforderlichen Konfigurationen vor. Weitere Informationen zur Filterung erhalten Sie unter the section called “Ereignisfilterung”.

   - Fügen Sie unter Filterkriterien Definitionen für Filterkriterien hinzu, um zu bestimmen, ob ein Ereignis verarbeitet werden soll oder nicht.

10. Nehmen Sie unter Fehlerbehandlung die erforderlichen Konfigurationen vor. Weitere Informationen zur Fehlerbehandlung finden Sie unter the section called “Fehlgeschlagene Aufrufe beibehalten”.

    - Geben Sie als On-failure Ziel den ARN Ihres Ziels bei einem Ausfall an.

11. Geben Sie unter Tags die Tags ein, die dieser Zuordnung von Ereignisquellen zugeordnet werden sollen.
12. Wählen Sie hinzufügen aus, um den Auslöser zu erstellen.

Sie können die Ereignisquellenzuordnung auch mithilfe der AWS CLI mit dem Befehl create-event-source-mapping erstellen. Im folgenden Beispiel wird eine Ereignisquellenzuordnung erstellt, um die Lambda-Ereignisfunktion `my-msk-function` ausgehend von der `LATEST` Nachricht dem `AWSKafkaTopic` Thema zuzuordnen. Dieser Befehl verwendet das SourceAccessConfigurationObjekt auch, um Lambda anzuweisen, bei der Verbindung mit dem Cluster die SASL/SCRAMAuthentifizierung zu verwenden.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Wenn der Cluster mTLS-Authentifizierung verwendet, fügen Sie ein SourceAccessConfigurationObjekt hinzu, das einen Secrets Manager Manager-Schlüssel-ARN spezifiziert`CLIENT_CERTIFICATE_TLS_AUTH`. Das wird im folgenden Befehl veranschaulicht:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Wenn der Cluster die IAM-Authentifizierung verwendet, benötigen Sie kein SourceAccessConfigurationObjekt. Das wird im folgenden Befehl veranschaulicht:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
```

```
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Erstellen von kontenübergreifenden Zuordnungen von Ereignisquellen in Lambda

Sie können private Konnektivität mit mehreren VPC verwenden, um eine Lambda-Ereignisfunktion mit einem bereitgestellten MSK-Cluster in einem anderen zu verbinden. AWS-Konto Multi-VPC Konnektivität verwendet AWS PrivateLink, wodurch der gesamte Datenverkehr im Netzwerk bleibt. AWS

> **Note**
>
> Sie können keine kontenübergreifenden Zuordnungen von Ereignisquellen für Serverless-MSK-Cluster erstellen.

Um eine kontenübergreifende Zuordnung von Ereignisquellen zu erstellen, müssen Sie zunächst die Multi-VPC-Konnektivität für den MSK-Cluster konfigurieren. Verwenden Sie beim Erstellen der Zuordnung von Ereignisquellen den ARN der verwalteten VPC-Verbindung anstelle des Cluster-ARNs, wie in den folgenden Beispielen gezeigt. Der CreateEventSourceMappingVorgang unterscheidet sich auch je nachdem, welchen Authentifizierungstyp der MSK-Cluster verwendet.

Example– Erstellen Sie eine kontoübergreifende Zuordnung von Ereignisquellen für einen Cluster, der die IAM-Authentifizierung verwendet

Wenn der Cluster die rollenbasierte IAM-Authentifizierung verwendet, benötigen Sie kein Objekt. SourceAccessConfiguration Beispiel:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example— Erstellen Sie eine kontenübergreifende Zuordnung von Ereignisquellen für einen Cluster, der Authentifizierung verwendet SASL/SCRAM

Wenn der Cluster SASL/SCRAM Authentifizierung verwendet, müssen Sie ein SourceAccessConfigurationObjekt angeben, das einen geheimen ARN von Secrets Manager spezifiziert`SASL_SCRAM_512_AUTH`.

Es gibt zwei Möglichkeiten, Geheimnisse für kontoübergreifende Amazon MSK-Ereignisquellenzuordnungen mit Authentifizierung zu verwenden: SASL/SCRAM

- Erstellen Sie ein Geheimnis im Lambda Event Function-Konto und synchronisieren Sie es mit dem Clustergeheimnis. Erstellen Sie eine Rotation, um die beiden Secrets synchron zu halten. Mit dieser Option haben Sie die Kontrolle über das Geheimnis des Funktionskontos.
- Verwenden Sie das Secret, das dem MSK-Cluster zugeordnet ist. Dieses Geheimnis muss kontoübergreifenden Zugriff auf das Lambda Event Function-Konto ermöglichen. Weitere Informationen finden Sie unter Berechtigungen für AWS Secrets Manager -Secrets für Benutzer in einem anderen Konto.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example– Erstellen Sie eine kontoübergreifende Zuordnung von Ereignisquellen für einen Cluster, der die mTLS-Authentifizierung verwendet

Wenn der Cluster mTLS-Authentifizierung verwendet, müssen Sie ein SourceAccessConfigurationObjekt angeben, das einen geheimen ARN von Secrets Manager spezifiziert`CLIENT_CERTIFICATE_TLS_AUTH`. Das Geheimnis kann im Clusterkonto oder im Lambda Event Function-Konto gespeichert werden.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
```

```
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
"arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Alle Amazon-MSK-Konfigurationsparameter von Ereignisquellen in Lambda

Alle Lambda-Ereignisquelltypen verwenden dieselben CreateEventSourceMappingUpdateEventSourceMappingAPI-Operationen. Allerdings gelten nur einige der Parameter für selbstverwaltetes Amazon MSK, wie in der folgenden Tabelle dargestellt.

Parameter	Erforderlich	Standard	Hinweise
AmazonManagedKafkaEventSourceConfig	N	Enthält das ConsumerGroupId Feld, das standardmäßig einen eindeutigen Wert hat.	Kann nur auf „Erstellen“ festgelegt werden
BatchSize	N	100	Höchstwert: 10 000.
DestinationConfig	N	N/A	the section called “Fehlgeschlagene Aufrufe beibehalten”
Aktiviert	N	Wahr	
BisectBatchOnFunctionError	N	Falsch	the section called “Versuchen Sie es erneut mit den Konfigurationen”
FunctionResponseTypes	N	N/A	the section called “Versuchen Sie es erneut mit den Konfigurationen”
MaximumRecordAgeInSeconds	N	-1 (unendlich)	the section called “Versuchen Sie

Parameter	Erforderlich	Standard	Hinweise
			es erneut mit den Konfigurationen”
MaximumRetryAttempts	N	-1 (unendlich)	the section called “Versuchen Sie es erneut mit den Konfigurationen”
EventSourceArn	Y	N/A	Kann nur auf „Erstellen“ festgelegt werden
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet
FunctionName	Y	N/A	
KMSKeyArn	N	N/A	the section called “Verschlüsselung der Filterkriterien”
MaximumBatchingWindowInSeconds	N	500 ms	Batching-Verhalten
ProvisionedPollersConfig	N	`MinimumPollers` : Standardwert von 1, wenn nicht angegeben `MaximumPollers` : Standardwert von 200, wenn nicht angegeben `PollerGroupName` : N/A	the section called “Modus bereitgestellter Kapazität”

Parameter	Erforderlich	Standard	Hinweise
SourceAccessConfigurations	N	Keine Anmeldedaten	SASL/SCRAM oder die Anmeldeinformationen CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) für Ihre Ereignisquelle
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON, oder LATEST Kann nur auf „Erstellen“ festgelegt werden
StartingPositionTimestamp	N	N/A	Erforderlich, wenn es auf StartingPosition AT_TIMESTAMP gesetzt ist
Tags (Markierungen)	N	N/A	the section called “Tags für die Zuordnung von Ereignisquellen”
Topics	Y	N/A	Kafka-Thema-Name Kann nur auf „Erstellen“ festgelegt werden

> **Note**
>
> Wenn Sie a angeben`PollerGroupName`, können sich mehrere ESMs innerhalb derselben Amazon VPC die Kapazität der Event Poller Unit (EPU) teilen. Sie können diese Option

verwenden, um die Kosten für Ihre ESMs im Bereitstellungsmodus zu optimieren. Anforderungen für die ESM-Gruppierung:

- ESMs müssen sich in derselben Amazon-VPC befinden
- Maximal 100 ESMs pro Befragungsgruppe
- Die Gesamtzahl der Abfragenden für alle ESMs in einer Gruppe darf 2000 nicht überschreiten

Sie können den aktualisieren`PollerGroupName`, um einen ESM in eine andere Gruppe zu verschieben, oder einen ESM aus einer Gruppe entfernen, indem Sie ihn `PollerGroupName` auf eine leere Zeichenfolge („“) setzen.

Tutorial: Verwenden einer Amazon MSK-Ereignisquellenzuordnung zum Aufrufen einer Lambda-Ereignisfunktion

In diesem Tutorial gehen Sie wie folgt vor:

- Erstellen Sie eine Lambda-Ereignisfunktion in demselben AWS Konto wie ein vorhandener Amazon MSK-Cluster.
- Konfigurieren Sie Netzwerk und Authentifizierung für Lambda für die Kommunikation mit Amazon MSK.
- Richten Sie eine Lambda Amazon MSK-Ereignisquellenzuordnung ein, die Ihre Funktion ausführt, wenn Ereignisse im Thema auftauchen.

Wenn Sie mit diesen Schritten fertig sind und Ereignisse an Amazon MSK gesendet werden, können Sie eine Funktion einrichten, um diese Ereignisse automatisch mit Ihrem eigenen benutzerdefinierten Lambda-Code zu verarbeiten.

Was können Sie mit diesem Feature machen?

Beispiellösung: Verwenden Sie eine MSK-Zuordnung von Ereignisquellen, um Ihren Kunden Live-Ergebnisse zu liefern.

Stellen Sie sich das folgende Szenario vor: Ihr Unternehmen hostet eine Webanwendung, mit der Ihre Kunden Informationen über Live-Ereignisse, z. B. Sportspiele, abrufen können. Aktuelle Informationen aus dem Spiel werden Ihrem Team über ein Kafka-Thema auf Amazon MSK zur Verfügung gestellt. Sie möchten eine Lösung entwerfen, die Aktualisierungen aus dem MSK-Thema

abruft, um den Kunden in einer von Ihnen entwickelten Anwendung eine aktualisierte Ansicht des Live-Ereignisses zu bieten. Sie haben sich für den folgenden Entwurfsansatz entschieden: Ihre Client-Anwendungen kommunizieren mit einem serverlosen Backend, das in gehostet wird. AWS Clients stellen mithilfe der Amazon WebSocket API Gateway eine Verbindung über Websocket-Sitzungen her.

Bei dieser Lösung benötigen Sie eine Komponente, die MSK-Ereignisse liest, eine benutzerdefinierte Logik ausführt, um diese Ereignisse für die Anwendungsschicht vorzubereiten und diese Informationen dann an die API-Gateway-API weiterleitet. Sie können diese Komponente implementieren AWS Lambda, indem Sie Ihre benutzerdefinierte Logik in einer Funktion bereitstellen und sie dann mit einer AWS Lambda Amazon MSK-Ereignisquellenzuordnung aufrufen.

Weitere Informationen zur Implementierung von Lösungen mit der Amazon API Gateway WebSocket API finden Sie WebSocket in den API-Tutorials in der API Gateway-Dokumentation.

Voraussetzungen

Ein AWS Konto mit den folgenden vorkonfigurierten Ressourcen:

Um diese Voraussetzungen zu erfüllen, empfehlen wir den Abschnitt Erste Schritte mit Amazon MSK in der Amazon MSK-Dokumentation.

- Ein Amazon-MSK-Cluster. Siehe Erstellen eines Amazon MSK-Clusters in Erste Schritte mit Amazon MSK.
- Die folgende Konfiguration:
  - Vergewissern Sie sich, dass die rollenbasierte IAM-Authentifizierung in den Sicherheitseinstellungen Ihres Clusters aktiviert ist. Dies verbessert Ihre Sicherheit, indem Sie Ihre Lambda-Ereignisfunktion so einschränken, dass sie nur auf die benötigten Amazon MSK-Ressourcen zugreift. Dies ist bei neuen Amazon MSK-Clustern standardmäßig aktiviert.
  - Vergewissern Sie sich, dass der öffentliche Zugang in den Netzwerkeinstellungen Ihres Clusters deaktiviert ist. Wenn Sie den Zugang Ihres Amazon MSK Clusters zum Internet einschränken, erhöht sich Ihre Sicherheit, da die Anzahl der Vermittler, die Ihre Daten verarbeiten, begrenzt wird. Dies ist bei neuen Amazon MSK-Clustern standardmäßig aktiviert.
- Ein Kafka-Thema in Ihrem Amazon MSK-Cluster, das Sie für diese Lösung verwenden können. Weitere Informationen finden Sie unter Erstellen eines Themas unter Erste Schritte mit Amazon MSK.
- Ein Kafka-Admin-Host, der so eingerichtet ist, dass er Informationen von Ihrem Kafka-Cluster abruft und Kafka-Ereignisse zu Testzwecken an Ihr Thema sendet, z. B. eine Amazon EC2-Instance mit

der Kafka-Admin-CLI und der installierten Amazon-MSK-IAM-Bibliothek. Siehe Erstellen eines Client-Rechners in Erste Schritte mit Amazon MSK.

Nachdem Sie diese Ressourcen eingerichtet haben, erfassen Sie die folgenden Informationen aus Ihrem AWS Konto, um zu bestätigen, dass Sie bereit sind, fortzufahren.

- Der Name Ihres Amazon MSK-Clusters. Sie finden diese Informationen in der Amazon-MSK-Konsole.
- Die Cluster-UUID, Teil des ARN für Ihren Amazon MSK-Cluster, den Sie in der Amazon MSK-Konsole finden können. Folgen Sie den Verfahren unter Cluster auflisten in der Amazon MSK-Dokumentation, um diese Informationen zu finden.
- Die mit Ihrem Amazon MSK-Cluster verbundenen Sicherheitsgruppen. Sie finden diese Informationen in der Amazon-MSK-Konsole. Bezeichnen Sie diese in den folgenden Schritten als Ihre*`clusterSecurityGroups`*.
- Die ID der Amazon VPC, die Ihren Amazon MSK-Cluster enthält. Sie können diese Informationen finden, indem Sie in der Amazon MSK-Konsole die mit Ihrem Amazon MSK-Cluster verbundenen Subnetze identifizieren und dann in der Amazon VPC-Konsole die mit dem Subnetz verbundene Amazon VPC identifizieren.
- Der Name des in Ihrer Lösung verwendeten Kafka-Themas. Sie können diese Informationen finden, indem Sie Ihren Amazon MSK-Cluster mit der Kafka `topics` CLI von Ihrem Kafka-Administrationshost aus aufrufen. Weitere Informationen zur Themen-CLI finden Sie in der Kafka-Dokumentation unter Themen hinzufügen und entfernen.
- Der Name einer Verbrauchergruppe für Ihr Kafka-Thema, geeignet für die Verwendung durch Ihre Lambda-Event-Funktion. Diese Gruppe kann automatisch von Lambda erstellt werden, Sie müssen sie also nicht mit der Kafka-CLI erstellen. Wenn Sie Ihre Verbrauchergruppen verwalten müssen, finden Sie weitere Informationen über die Verbrauchergruppen-CLI unter Verwalten von Verbrauchergruppen in der Kafka-Dokumentation.

Die folgenden Berechtigungen in Ihrem AWS Konto:

- Erlaubnis, eine Funktion zu erstellen und zu verwalten.
- Erlaubnis, IAM-Richtlinien zu erstellen und diese Ihrer Funktion zuzuordnen.
- Berechtigung zum Erstellen von Amazon VPC-Endpunkten und zum Ändern der Netzwerkkonfiguration in der Amazon VPC, die Ihren Amazon MSK-Cluster hostet.

Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

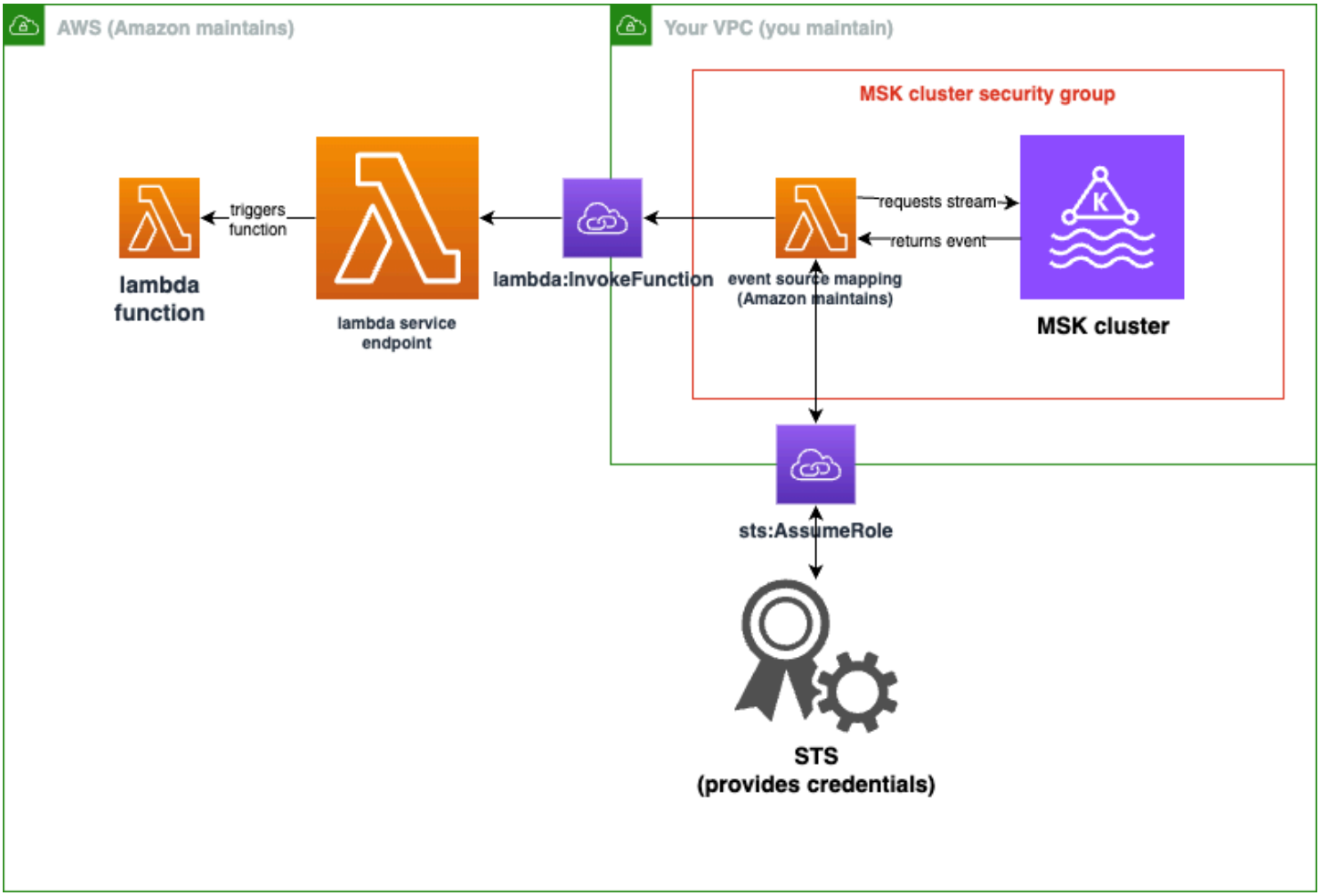
Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Konfigurieren Sie die Netzwerkkonnektivität für Lambda zur Kommunikation mit Amazon MSK

Wird verwendet AWS PrivateLink , um Lambda und Amazon MSK zu verbinden. Sie können dies tun, indem Sie Amazon VPC-Endpunkte in der Amazon VPC-Konsole erstellen. Weitere Informationen zur Netzwerkkonfiguration finden Sie unter the section called “Einrichtung von Cluster und Netzwerk”.

Wenn eine Amazon MSK-Ereignisquellenzuordnung im Namen einer Lambda-Ereignisfunktion ausgeführt wird, übernimmt sie die Ausführungsrolle der Funktion. Diese IAM-Rolle autorisiert die Zuordnung für den Zugriff auf durch IAM gesicherte Ressourcen, wie z. B. Ihren Amazon MSK-Cluster. Obwohl sich die Komponenten eine gemeinsame Ausführungsrolle teilen, haben die Amazon MSK-Zuordnung und Ihre Funktion unterschiedliche Konnektivitätsanforderungen für ihre jeweiligen Aufgaben, wie in der folgenden Abbildung dargestellt.



Ihre Zuordnung von Ereignisquellen gehört zu Ihrer Amazon MSK-Cluster-Sicherheitsgruppe. In diesem Vernetzungsschritt erstellen Sie Amazon VPC-Endpunkte von Ihrer Amazon MSK-Cluster-VPC, um die Zuordnung von Ereignisquellen mit den Lambda- und STS-Services zu verbinden. Sichern Sie diese Endpunkte, damit sie Datenverkehr von Ihrer Amazon MSK-Cluster-Sicherheitsgruppe akzeptieren. Passen Sie dann die Sicherheitsgruppen des Amazon MSK-Clusters an, damit die Zuordnung von Ereignisquellen mit dem Amazon MSK-Cluster kommunizieren kann.

Sie können die folgenden Schritte mit dem AWS-Managementkonsole.

So konfigurieren Sie Schnittstellen-Amazon-VPC-Endpunkte, um Lambda und Amazon MSK zu verbinden

1. Erstellen Sie eine Sicherheitsgruppe für Ihre Schnittstelle Amazon VPC-Endpunkte,*`endpointSecurityGroup`*, die eingehenden TCP-Verkehr auf 443 von ermöglicht. *`clusterSecurityGroups`* Befolgen Sie das Verfahren unter Erstellen einer Sicherheitsgruppe in der Amazon EC2-Dokumentation, um eine Sicherheitsgruppe zu erstellen. Folgen Sie dann

dem Verfahren unter Hinzufügen von Regeln zu einer Sicherheitsgruppe in der Amazon EC2-Dokumentation, um entsprechende Regeln hinzuzufügen.

Erstellen Sie eine Sicherheitsgruppe mit den folgenden Informationen:

Wenn Sie Ihre Regeln für eingehenden Datenverkehr hinzufügen, erstellen Sie eine Regel für jede Sicherheitsgruppe in. *clusterSecurityGroups* Für jede Regel:

- Wählen Sie für Type (Typ) HTTPS aus.
- Wählen Sie für Quelle eine von. *clusterSecurityGroups*

2. Erstellen Sie einen Endpunkt, der den Lambda-Service mit der Amazon VPC verbindet, die Ihren Amazon MSK-Cluster enthält. Folgen Sie dem Verfahren unter Schnittstellenendpunkt erstellen.

   Erstellen Sie einen Schnittstellenendpunkt mit den folgenden Informationen:

   - Wählen Sie als Dienstname aus`com.amazonaws.`*regionName*`.lambda`, wo Ihre Funktion *regionName* gehostet wird.
   - Wählen Sie für VPC die Amazon VPC aus, die Ihren Amazon MSK-Cluster enthält.
   - Wählen Sie für Sicherheitsgruppen die Gruppen aus*endpointSecurityGroup*, die Sie zuvor erstellt haben.
   - Wählen Sie für Subnetze die Subnetze aus, die Ihren Amazon MSK-Cluster hosten.
   - Geben Sie für die Richtlinie das folgende Richtliniendokument an, das den Endpunkt für die Verwendung durch den Lambda-Serviceprinzipal für die `lambda:InvokeFunction`-Aktion sichert.

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

- Stellen Sie sicher, dass DNS-Name aktivieren weiterhin aktiviert ist.

3. Erstellen Sie einen Endpunkt, der den AWS STS Service mit der Amazon VPC verbindet, die Ihren Amazon MSK-Cluster enthält. Folgen Sie dem Verfahren unter Schnittstellenendpunkt erstellen.

   Erstellen Sie einen Schnittstellenendpunkt mit den folgenden Informationen:

   - Wählen Sie als Dienstname die Option aus. AWS STS
   - Wählen Sie für VPC die Amazon VPC aus, die Ihren Amazon MSK-Cluster enthält.
   - Wählen Sie für Sicherheitsgruppen die Option aus*endpointSecurityGroup*.
   - Wählen Sie für Subnetze die Subnetze aus, die Ihren Amazon MSK-Cluster hosten.
   - Geben Sie für die Richtlinie das folgende Richtliniendokument an, das den Endpunkt für die Verwendung durch den Lambda-Serviceprinzipal für die `sts:AssumeRole`-Aktion sichert.

     ```
     {
         "Statement": [
             {
                 "Action": "sts:AssumeRole",
                 "Effect": "Allow",
                 "Principal": {
                     "Service": [
                         "lambda.amazonaws.com"
                     ]
                 },
                 "Resource": "*"
             }
         ]
     }
     ```

   - Stellen Sie sicher, dass DNS-Name aktivieren weiterhin aktiviert ist.

4. Lassen Sie für jede Sicherheitsgruppe, die Ihrem Amazon MSK-Cluster zugeordnet ist, d. h. in*clusterSecurityGroups*, Folgendes zu:

   - Erlauben Sie allen eingehenden und ausgehenden TCP-Verkehr auf 9098 für alle*clusterSecurityGroups*, auch für sich selbst.
   - Den gesamten ausgehenden TCP-Verkehr auf 443 zulassen.

Ein Teil dieses Datenverkehrs wird durch die Standardregeln der Sicherheitsgruppe zugelassen. Wenn Ihr Cluster also nur einer einzigen Sicherheitsgruppe angehört und diese Gruppe über Standardregeln verfügt, sind zusätzliche Regeln nicht erforderlich. Um die Regeln für Sicherheitsgruppen anzupassen, befolgen Sie die Verfahren unter Hinzufügen von Regeln zu einer Sicherheitsgruppe in der Amazon EC2-Dokumentation.

Fügen Sie Ihren Sicherheitsgruppen Regeln mit den folgenden Informationen hinzu:

- Geben Sie für jede eingehende oder ausgehende Regel für Port 9098 Folgendes an
  - Wählen Sie für Type (Typ) Custom TCP (Benutzerdefiniertes TCP).
  - Geben Sie als Portbereich 9098 an.
  - Geben Sie als Quelle einen von an. *clusterSecurityGroups*
- Wählen Sie für jede eingehende Regel für Port 443 als Typ die Option HTTPS aus.

Erstellen Sie eine IAM-Rolle, die Lambda aus Ihrem Amazon-MSK-Thema lesen kann

Identifizieren Sie die Authentifizierungsanforderungen für Lambda, um aus Ihrem Amazon MSK-Thema zu lesen und definieren Sie sie dann in einer Richtlinie. Erstellen Sie eine Rolle*lambdaAuthRole*, die Lambda autorisiert, diese Berechtigungen zu verwenden. Autorisieren Sie Aktionen in Ihrem Amazon MSK-Cluster mithilfe von `kafka-cluster` IAM-Aktionen. Autorisieren Sie Lambda anschließend, Amazon MSK `kafka` - und Amazon EC2 EC2-Aktionen durchzuführen, die für die Erkennung Ihres Amazon MSK-Clusters und die Verbindung zu ihm erforderlich sind, sowie CloudWatch Aktionen, damit Lambda protokollieren kann, was es getan hat.

So beschreiben Sie die Authentifizierungsanforderungen für Lambda, um von Amazon MSK zu lesen

1. Schreiben Sie ein IAM-Richtliniendokument (ein JSON-Dokument), das es Lambda ermöglicht*clusterAuthPolicy*, mithilfe Ihrer Kafka-Verbrauchergruppe aus Ihrem Kafka-Thema in Ihrem Amazon MSK-Cluster zu lesen. Für Lambda muss beim Lesen eine Kafka-Verbrauchergruppe festgelegt werden.

   Ändern Sie die folgende Vorlage, um sie an Ihre Voraussetzungen anzupassen:

   JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:us-
east-1:111122223333:cluster/mskClusterName/cluster-uuid",
                "arn:aws:kafka:us-
east-1:111122223333:topic/mskClusterName/cluster-uuid/mskTopicName",
                "arn:aws:kafka:us-
east-1:111122223333:group/mskClusterName/cluster-uuid/mskGroupName"
            ]
        }
    ]
}
```

Weitere Informationen erhalten Sie von the section called “Konfigurieren von -Berechtigungen”. Wenn Sie Ihre Richtlinie schreiben:

- Ersetzen Sie *us-east-1* und *111122223333* durch das AWS-Region und AWS-Konto Ihres Amazon MSK-Clusters.
- Geben Sie für *mskClusterName* den Namen Ihres Amazon MSK-Clusters an.
- Geben Sie für *cluster-uuid* die UUID im ARN für Ihren Amazon MSK-Cluster an.
- Geben Sie für *mskTopicName* den Namen Ihres Kafka-Themas an.
- Geben Sie für *mskGroupName* den Namen Ihrer Kafka-Verbrauchergruppe an.

2. Identifizieren Sie Amazon MSK, Amazon EC2 und die CloudWatch Berechtigungen, die Lambda benötigt, um Ihren Amazon MSK-Cluster zu erkennen und zu verbinden, und protokollieren Sie diese Ereignisse.

   Die `AWSLambdaMSKExecutionRole`-verwaltete Richtlinie definiert die erforderlichen Berechtigungen auf unzulässige Weise. Verwenden Sie es in den folgenden Schritten.

Beurteilen Sie in einer Produktionsumgebung `AWSLambdaMSKExecutionRole`, um Ihre Ausführungsrollenrichtlinie auf der Grundlage des Prinzips der geringsten Berechtigung einzuschränken und schreiben Sie dann eine Richtlinie für Ihre Rolle, die diese verwaltete Richtlinie ersetzt.

Weitere Informationen zur IAM-Richtliniensprache finden Sie unter IAM-Dokumentation.

Nachdem Sie nun Ihr Richtliniendokument verfasst haben, erstellen Sie eine IAM-Richtlinie, die Sie mit Ihrer Rolle verknüpfen können. Sie können dies über die Konsole mit dem folgenden Verfahren tun.

So erstellen Sie eine IAM-Richtlinie aus Ihrem Richtliniendokument

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die IAM-Konsole unter. https://console.aws.amazon.com/iam/
2. Wählen Sie im Navigationsbereich auf der linken Seite Policies (Richtlinien).
3. Wählen Sie Richtlinie erstellen aus.
4. Wählen Sie im Bereich Policy editor (Richtlinien-Editor) die Option JSON aus.
5. Einfügen*`clusterAuthPolicy`*.
6. Wenn Sie mit dem Hinzufügen von Berechtigungen zur Richtlinie fertig sind, wählen Sie Next (Weiter) aus.
7. Geben Sie auf der Seite Review and create (Überprüfen und erstellen) unter Name einen Namen und unter Description (Beschreibung) (optional) eine Beschreibung für die Richtlinie ein, die Sie erstellen. Überprüfen Sie Permissions defined in this policy (In dieser Richtlinie definierte Berechtigungen), um die Berechtigungen einzusehen, die von Ihrer Richtlinie gewährt werden.
8. Wählen Sie Create policy (Richtlinie erstellen) aus, um Ihre neue Richtlinie zu speichern.

Weitere Informationen finden Sie unter Erstellen von IAM-Richtlinien in der IAM-Dokumentation.

Nachdem Sie nun über die entsprechenden IAM-Richtlinien verfügen, erstellen Sie eine Rolle und ordnen Sie sie dieser zu. Sie können dies über die Konsole mit dem folgenden Verfahren tun.

So erstellen Sie eine Ausführungsrolle in der IAM-Konsole

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.

3. Wählen Sie unter Typ der vertrauenswürdigen Entität die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall Lambda aus.
5. Wählen Sie Weiter aus.
6. Wählen Sie die folgenden Richtlinien:
   - *clusterAuthPolicy*
   - AWSLambdaMSKExecutionRole
7. Wählen Sie Weiter aus.
8. Geben Sie als Rollenname den Text Rolle erstellen ein ***lambdaAuthRole*** und wählen Sie dann Create role aus.

Weitere Informationen finden Sie unter the section called “Ausführungsrolle (Berechtigungen für Funktionen zum Zugriff auf andere Ressourcen)”.

Erstellen Sie eine Lambda-Ereignisfunktion zum Lesen aus Ihrem Amazon MSK-Thema

Erstellen Sie eine Lambda-Ereignisfunktion, die für die Verwendung Ihrer IAM-Rolle konfiguriert ist. Sie können Ihre Funktion mithilfe der Konsole erstellen.

Um eine Funktion mit Ihrer Authentifizierungskonfiguration zu erstellen

1. Öffnen Sie die Lambda-Konsole und wählen Sie in der Kopfzeile Create Event Function aus.
2. Wählen Sie Verfassen von Grund auf aus.
3. Geben Sie als Funktionsnamen einen geeigneten Namen Ihrer Wahl an.
4. Wählen Sie für Laufzeit die neueste unterstützte Version von `Node.js`, um den in diesem Tutorial bereitgestellten Code zu verwenden.
5. Wählen Sie Standardeausführungsrolle ändern aus.
6. Wählen Sie Eine vorhandene Rolle verwenden aus.
7. Wählen Sie für Existierende Rolle die Option aus*lambdaAuthRole*.

In einer Produktionsumgebung müssen Sie in der Regel weitere Richtlinien zur Ausführungsrolle für Ihre Lambda-Ereignisfunktion hinzufügen, um Ihre Amazon MSK-Ereignisse sinnvoll verarbeiten zu können. Weitere Informationen zum Hinzufügen von Richtlinien zu Ihrer Rolle finden Sie unter Hinzufügen oder Entfernen von Identitätsberechtigungen in der IAM-Dokumentation.

Erstellen Sie eine Ereignisquellenzuordnung zu Ihrer Lambda-Ereignisfunktion

Ihre Amazon MSK-Zuordnung von Ereignisquellen liefert dem Lambda-Service die notwendigen Informationen, um Ihr Lambda aufzurufen, wenn entsprechende Amazon MSK-Ereignisse auftreten. Sie können über die Konsole eine Amazon MSK-Zuordnung erstellen. Erstellen Sie einen Lambda-Trigger, dann wird die Zuordnung von Ereignisquellen automatisch eingerichtet.

So erstellen Sie einen Lambda-Trigger (und eine Zuordnung von Ereignisquellen)

1. Navigieren Sie zur Übersichtsseite Ihrer Lambda Event Function.
2. Wählen Sie im Abschnitt „Übersicht über die Ereignisfunktion“ unten links die Option Auslöser hinzufügen aus.
3. Wählen Sie in der Dropdownliste Quelle auswählen die Option Amazon MSK aus.
4. Legen Sie keine Authentifizierung fest.
5. Wählen Sie für MSK-Cluster den Namen Ihres Clusters aus.
6. Geben Sie für Batch size (Stapelgröße) 1 ein. Dieser Schritt erleichtert das Testen dieses Features, ist aber für die Produktion nicht ideal.
7. Geben Sie für Themenname den Namen Ihres Kafka-Themas ein.
8. Geben Sie unter Verbrauchergruppen-ID die ID Ihrer Kafka-Verbrauchergruppe an.

Aktualisieren Sie Ihre Lambda-Event-Funktion, um Ihre Streaming-Daten zu lesen

Lambda liefert Informationen über Kafka-Ereignisse über den Parameter der Ereignismethode. Ein Beispiel für die Struktur eines Amazon MSK-Ereignisses finden Sie unter the section called “Beispielereignis”. Nachdem Sie verstanden haben, wie Lambda-weitergeleitete Amazon MSK-Ereignisse interpretiert werden, können Sie Ihren Lambda-Ereignisfunktionscode ändern, um die darin enthaltenen Informationen zu verwenden.

Geben Sie den folgenden Code für Ihre Lambda-Ereignisfunktion ein, um den Inhalt eines Lambda Amazon MSK-Ereignisses zu Testzwecken zu protokollieren:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von .NET.

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);
```

```
                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Go.

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
```

```
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);
```

```
                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Amazon MSK-Ereignis mit Lambda verwenden. JavaScript

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

Ein Amazon MSK-Ereignis mit Lambda verwenden. TypeScript

```
import { MSKEvent, Context } from "aws-lambda";
```

```
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Python.

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Ruby.

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon-MSK-Ereignisses mit Lambda unter Verwendung von Rust.

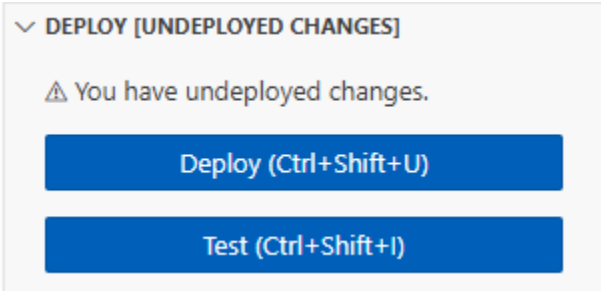
```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
```

```
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
           }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

Sie können Ihrem Lambda Funktionscode über die Konsole zur Verfügung stellen.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.
4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Testen Sie Ihre Lambda-Ereignisfunktion, um sicherzustellen, dass sie mit Ihrem Amazon MSK-Thema verbunden ist

Sie können jetzt überprüfen, ob Ihr Lambda von der Ereignisquelle aufgerufen wird, indem Sie die Ereignisprotokolle überprüfen CloudWatch .

Um zu überprüfen, ob Ihre Lambda-Ereignisfunktion aufgerufen wird

1. Verwenden Sie Ihren Kafka-Admin-Host, um Kafka-Ereignisse mithilfe der `kafka-console-producer`-CLI zu generieren. Weitere Informationen finden Sie in der Kafka-Dokumentation unter Einige Ereignisse in das Thema schreiben. Senden Sie genügend Ereignisse, um den Stapel zu füllen, der durch die Batchgröße für Ihre im vorherigen Schritt definierte Ereignisquellenzuordnung definiert wurde, oder Lambda wartet, bis weitere Informationen aufgerufen werden.
2. Wenn Ihre Funktion ausgeführt wird, schreibt Lambda, was passiert ist. CloudWatch Navigieren Sie in der Konsole zur Detailseite Ihrer Lambda Event Function.
3. Wählen Sie die Registerkarte Configuration aus.
4. Wählen Sie in der Seitenleiste die Option Überwachungs- und Betriebstools.
5. Identifizieren Sie die CloudWatch Protokollgruppe unter Logging-Konfiguration. Die Protokollgruppe sollte mit `/aws/lambda` beginnen. Wählen Sie den Link zur Protokollgruppe.
6. Suchen Sie in der CloudWatch Konsole in den Protokollereignissen nach den Protokollereignissen, die Lambda an den Protokollstream gesendet hat. Stellen Sie fest, ob es Protokollereignisse gibt, die die Nachricht aus Ihrem Kafka-Ereignis enthalten, wie in der folgenden Abbildung dargestellt. Falls ja, haben Sie erfolgreich eine Lambda-Ereignisfunktion mit Amazon MSK mit einer Lambda-Ereignisquellenzuordnung verbunden.



Verwendung von Lambda mit selbstverwaltetem Apache Kafka

In diesem Thema wird die Verwendung von Lambda mit einem selbstverwalteten Kafka-Cluster beschrieben. In der AWS Terminologie umfasst ein selbstverwalteter Cluster auch nicht AWS gehostete Kafka-Cluster. Sie können Ihren Kafka-Cluster beispielsweise bei einem Cloud-Anbieter wie Confluent Cloud oder Redpanda hosten.

In diesem Kapitel wird erklärt, wie Sie einen selbstverwalteten Apache Kafka-Cluster als Ereignisquelle für Ihre Lambda Event Function verwenden. Der allgemeine Prozess zur Integration von selbstverwaltetem Apache Kafka mit Lambda umfasst die folgenden Schritte:

1. Cluster- und Netzwerkeinrichtung – Richten Sie zunächst Ihren selbstverwalteten Apache-Kafka-Cluster mit der richtigen Netzwerkkonfiguration ein, damit Lambda auf Ihren Cluster zugreifen kann.
2. Einrichtung der Zuordnung von Ereignisquellen – Erstellen Sie anschließend die Ressource der Zuordnung von Ereignisquellen, die Lambda benötigt, um Ihren Apache-Kafka-Cluster sicher mit Ihrer Funktion zu verbinden.
3. Einrichtung der Ereignisfunktion und der Berechtigungen — Stellen Sie abschließend sicher, dass Ihre Funktion korrekt eingerichtet ist und über die erforderlichen Berechtigungen für ihre Ausführungsrolle verfügt.

Apache Kafka als Ereignisquelle funktioniert ähnlich wie die Verwendung von Amazon Simple Queue Service (Amazon SQS) oder Amazon Kinesis. Lambda fragt intern nach neuen Nachrichten von der Ereignisquelle ab und ruft dann synchron die Lambda-Zielfunktion auf. Lambda liest die Nachrichten in Batches und stellt diese Ihrer Funktion als Ereignisnutzlast zur Verfügung. Die maximale Batchgröße ist konfigurierbar (Standardeinstellung: 100 Nachrichten). Weitere Informationen finden Sie unter the section called “Batching-Verhalten”.

Um den Durchsatz Ihrer selbstverwalteten Apache Kafka-Zuordnungen von Ereignisquellen zu optimieren, konfigurieren Sie den Bereitstellungsmodus. Im Bereitstellungsmodus können Sie die minimale und maximale Anzahl von Event-Pollern definieren, die Zuordnung von Ereignisquellen zugewiesen werden. Dies kann die Fähigkeit Ihrer Zuordnung von Ereignisquellen verbessern, unerwartete Nachrichtenspitzen zu bewältigen. Weitere Informationen finden Sie unter Bereitstellungsmodus.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Für Kafka-based Ereignisquellen unterstützt Lambda die Verarbeitung von Steuerungsparametern wie Batching-Fenstern und Batchgröße. Weitere Informationen finden Sie unter Batching-Verhalten.

Ein Beispiel für die Verwendung von selbstverwaltetem Kafka als Ereignisquelle finden Sie im Compute-Blog unter Verwenden von selbst gehostetem Apache Kafka als Ereignisquelle für. AWS Lambda AWS

Themen

- Beispielereignis
- Konfiguration Ihres selbstverwalteten Apache-Kafka-Clusters und -Netzwerks für Lambda
- Konfigurieren von Rollenberechtigungen für Lambda-Ausführungen
- Konfigurieren von selbstverwalteten Apache Kafka-Ereignisquellen für Lambda

Beispielereignis

Lambda sendet den Nachrichtenstapel im Event-Parameter, wenn es Ihre Lambda-Event-Funktion aufruft. Die Ereignisquelle enthält ein Array von Nachrichten. Jedes Array-Element enthält Details zum Kafka-Thema und Kafka-Partitions-ID, zusammen mit einem Zeitstempel und einer base64-codierten Nachricht.

```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
```

```
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

Konfiguration Ihres selbstverwalteten Apache-Kafka-Clusters und -Netzwerks für Lambda

Um Ihre Lambda Event Function mit Ihrem selbstverwalteten Apache Kafka-Cluster zu verbinden, müssen Sie Ihren Cluster und das Netzwerk, in dem er sich befindet, korrekt konfigurieren. Auf dieser Seite wird beschrieben, wie Sie Ihren Cluster und Ihr Netzwerk konfigurieren. Wenn Ihr Cluster und Ihr Netzwerk bereits ordnungsgemäß konfiguriert sind, lesen Sie unter the section called “Konfigurieren der Ereignisquelle” nach, wie Sie die Zuordnung von Ereignisquellen konfigurieren.

Themen

- Self-managed Einrichtung des Apache Kafka-Clusters
- Konfigurieren der Netzwerksicherheit

Self-managed Einrichtung des Apache Kafka-Clusters

Sie können Ihren selbstverwalteten Apache-Kafka-Cluster bei Cloud-Anbietern wie Confluent Cloud oder Redpanda hosten oder ihn in Ihrer eigenen Infrastruktur ausführen. Stellen Sie sicher, dass Ihr Cluster ordnungsgemäß konfiguriert ist und von dem Netzwerk aus zugänglich ist, über das Ihre Lambda-Ereignisquellenzuordnung eine Verbindung herstellt.

Konfigurieren der Netzwerksicherheit

Um Lambda vollen Zugriff auf selbstverwaltetes Apache Kafka über Ihre Zuordnung von Ereignisquellen zu ermöglichen, muss Ihr Cluster entweder einen öffentlichen Endpunkt (öffentliche IP-Adresse) verwenden oder Sie müssen Zugriff auf die Amazon VPC gewähren, in der Sie den Cluster erstellt haben.

Wenn Sie selbstverwaltetes Apache Kafka mit Lambda verwenden, erstellen Sie AWS PrivateLink VPC-Endpunkte, die Ihrer Event Function Zugriff auf die Ressourcen in Ihrer Amazon VPC gewähren.

> **Note**
>
> AWS PrivateLink VPC-Endpunkte sind für Funktionen mit Ereignisquellenzuordnungen erforderlich, die den Standardmodus (auf Abruf) für Ereignisabfragen verwenden. Wenn Ihre Ereignisquellenzuordnung den Bereitstellungsmodus verwendet, müssen Sie keine AWS PrivateLink VPC-Endpunkte konfigurieren.

Erstellen Sie einen Endpunkt, der den Zugriff auf die folgenden Ressourcen ermöglicht:

- Lambda – Erstellen Sie einen Endpunkt für den Lambda-Serviceprinzipal.
- AWS STS — Erstellen Sie einen Endpunkt für den, damit ein Service Principal eine Rolle AWS STS in Ihrem Namen übernehmen kann.
- Secrets Manager – Wenn Ihr Cluster Secrets Manager zum Speichern von Anmeldeinformationen verwendet, erstellen Sie einen Endpunkt für Secrets Manager.

Alternativ konfigurieren Sie ein NAT-Gateway auf jedem öffentlichen Subnetz in der Amazon-VPC. Weitere Informationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn Sie eine Zuordnung von Ereignisquellen für selbstverwaltetes Apache Kafka erstellen, prüft Lambda, ob Elastic-Network-Schnittstelle (ENI) für die für Ihre Amazon VPC konfigurierten Subnetze und Sicherheitsgruppen bereits vorhanden sind. Findet Lambda vorhandene ENIs, versucht es,

diese wiederzuverwenden. Andernfalls erstellt Lambda neue ENIs, um eine Verbindung mit der Ereignisquelle herzustellen und Ihre Funktion aufzurufen.

> **Note**
>
> Lambda-Ereignisfunktionen werden immer in VPCs ausgeführt, die dem Lambda-Service gehören. Die VPC-Konfiguration Ihrer Funktion hat keinen Einfluss auf die Zuordnung von Ereignisquellen. Nur die Netzwerkkonfiguration der Ereignisquelle bestimmt, wie Lambda sich mit Ihrer Ereignisquelle verbindet.

Konfigurieren Sie die Sicherheitsgruppen für die Amazon VPC, die Ihren Cluster enthält. Standardmäßig verwendet der selbstverwaltete Apache Kafka die folgenden Ports: `9092`.

- Eingehende Regeln – Erlauben Sie den gesamten Datenverkehr auf dem Standard-Broker-Port für die Sicherheitsgruppe, die mit Ihrer Ereignisquelle verbunden ist. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr — Erlauben Sie den gesamten Datenverkehr auf dem Port `443` für externe Ziele, wenn Ihre Funktion mit Diensten kommunizieren muss. AWS Alternativ können Sie auch eine selbstreferenzierende Sicherheitsgruppenregel verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen Diensten kommunizieren müssen. AWS
- Regeln für eingehenden Datenverkehr auf Amazon-VPC-Endpunkten – Wenn Sie einen Amazon-VPC-Endpunkt verwenden, muss die Sicherheitsgruppe, die mit Ihrem Amazon-VPC-Endpunkt verbunden ist, eingehenden Verkehr auf Port `443` von der Cluster-Sicherheitsgruppe zulassen.

Wenn Ihr Cluster eine Authentifizierung verwendet, können Sie auch die Endpunktrichtlinie für den Secrets-Manager-Endpunkt einschränken. Für den Aufruf der Secrets-Manager-API verwendet Lambda Ihre Funktionsrolle und nicht den Lambda-Serviceprinzipal.

Example VPC-Endpunktrichtlinie – Secrets Manager-Endpunkt

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
```

```
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

Wenn Sie Amazon VPC-Endpunkte verwenden, AWS leitet Ihre API-Aufrufe zum Aufrufen Ihrer Funktion über das Elastic Network Interface (ENI) des Endpunkts weiter. Der Prinzipal des Lambda-Dienstes muss `lambda:InvokeFunction` für alle Rollen und Funktionen aufrufen, die diese ENI verwenden.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Um sicherzustellen, dass Ihre Ereignisquellenzuordnung Ihre Lambda-Ereignisfunktion aufrufen kann, muss die VPC-Endpunktrichtlinie dem Lambda-Serviceprinzipal erlauben, und aufzurufen. `sts:AssumeRole` `lambda:InvokeFunction` Wenn Sie Ihre VPC-Endpunktrichtlinien so einschränken, dass nur API-Aufrufe zugelassen werden, die aus Ihrem Unternehmen stammen, kann die Zuordnung von Ereignisquellen nicht richtig funktionieren, weshalb in diesen Richtlinien `"Resource": "*"` erforderlich ist.

Die folgenden Beispiel-VPC-Endpunktrichtlinien zeigen, wie der erforderliche Zugriff auf den Lambda-Serviceprinzipal für die AWS STS - und Lambda-Endpunkte gewährt wird.

Example VPC-Endpunktrichtlinie — AWS STS Endpunkt

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
```

```
            "Resource": "*"
        }
    ]
}
```

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Konfigurieren von Rollenberechtigungen für Lambda-Ausführungen

Zusätzlich zum Zugriff auf Ihren selbstverwalteten Kafka-Cluster benötigt Ihre Lambda Event Function Berechtigungen, um verschiedene API-Aktionen auszuführen. Sie fügen diese Berechtigungen zur Ausführungsrolle der Funktion hinzu. Wenn Ihre Benutzer Zugriff auf API-Aktionen benötigen, fügen Sie der Identitätsrichtlinie für den AWS Identity and Access Management (IAM) -Benutzer oder die Rolle die erforderlichen Berechtigungen hinzu.

Themen

- Erforderliche Berechtigungen für die Lambda-Ereignisfunktion
- Optionale Berechtigungen für Lambda-Ereignisfunktionen
- Hinzufügen von Berechtigungen zu Ihrer Ausführungsrolle
- Gewähren von Benutzerzugriff mit einer IAM-Richtlinie

Erforderliche Berechtigungen für die Lambda-Ereignisfunktion

Um Protokolle in einer Protokollgruppe in Amazon CloudWatch Logs zu erstellen und zu speichern, muss Ihre Lambda-Ereignisfunktion in ihrer Ausführungsrolle über die folgenden Berechtigungen verfügen:

- Protokolle: CreateLogGroup
- Protokolle: CreateLogStream
- Protokolle: PutLogEvents

Optionale Berechtigungen für Lambda-Ereignisfunktionen

Ihre Lambda-Ereignisfunktion benötigt möglicherweise auch Berechtigungen für:

- Beschreiben Ihres Secrets-Manager-Secrets
- Greifen Sie auf Ihren AWS Key Management Service (AWS KMS) vom Kunden verwalteten Schlüssel zu.
- Zugriff auf Ihre Amazon VPC
- Senden von Datensätzen zu fehlgeschlagenen Aufrufen an ein Ziel

Secrets Manager und AWS KMS Berechtigungen

Abhängig von der Art der Zugriffskontrolle, die Sie für Ihre Kafka-Broker konfigurieren, benötigt Ihre Lambda Event Function möglicherweise die Erlaubnis, auf Ihr Secrets Manager-Geheimnis zuzugreifen oder Ihren vom AWS KMS Kunden verwalteten Schlüssel zu entschlüsseln. Um auf diese Ressourcen zuzugreifen, muss die Ausführungsrolle Ihrer Funktion die folgenden Berechtigungen besitzen:

- Secretsmanager: GetSecretValue
- kms:Decrypt

VPC-Berechtigungen

Wenn nur Benutzer innerhalb einer VPC auf Ihren selbstverwalteten Apache Kafka-Cluster zugreifen können, muss Ihre Lambda Event Function über die Berechtigung verfügen, auf Ihre Amazon VPC-Ressourcen zuzugreifen. Zu diesen Ressourcen gehören Ihre VPC, Subnetze, Sicherheitsgruppen

und Netzwerkschnittstellen. Um auf diese Ressourcen zuzugreifen, muss die Ausführungsrolle Ihrer Funktion die folgenden Berechtigungen besitzen:

- ec2: CreateNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeVpcs
- ec2: DeleteNetworkInterface
- ec2: DescribeSubnets
- ec2: DescribeSecurityGroups

Hinzufügen von Berechtigungen zu Ihrer Ausführungsrolle

Um auf andere AWS Dienste zuzugreifen, die Ihr selbstverwalteter Apache Kafka-Cluster verwendet, verwendet Lambda die Berechtigungsrichtlinien, die Sie in der Ausführungsrolle Ihrer Lambda Event Function definieren.

Standardmäßig ist es Lambda nicht gestattet, die erforderlichen oder optionalen Aktionen für einen selbstverwalteten Apache-Kafka-Cluster durchzuführen. Sie müssen diese Aktionen in einer IAM-Vertrauensrichtlinie für Ihre Ausführungsrolle erstellen und definieren. In diesem Beispiel wird gezeigt, wie Sie eine Richtlinie erstellen können, die Lambda den Zugriff auf Ihre Amazon-VPC-Ressourcen erlaubt.

JSON

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
```

```
            }
        ]
    }
```

Gewähren von Benutzerzugriff mit einer IAM-Richtlinie

Standardmäßig haben Benutzer und Rollen keine Berechtigung zum Ausführen von Ereignisquellen-API-Vorgängen. Um Benutzern in Ihrem Unternehmen oder Ihrem Konto Zugriff zu gewähren, erstellen oder aktualisieren Sie eine identitätsbasierte Richtlinie. Weitere Informationen finden Sie unter Steuern des Zugriffs auf AWS Ressourcen mithilfe von Richtlinien im IAM-Benutzerhandbuch.

Informationen zur Behebung von Authentifizierungs- und Autorisierungsfehlern finden Sie unter the section called “Fehlerbehebung”.

Konfigurieren von selbstverwalteten Apache Kafka-Ereignisquellen für Lambda

Um einen selbstverwalteten Apache Kafka-Cluster als Ereignisquelle für Ihre Lambda-Event-Funktion zu verwenden, erstellen Sie eine Zuordnung der Ereignisquelle das verbindet die beiden Ressourcen. Auf dieser Seite wird beschrieben, wie Sie eine Zuordnung von Ereignisquellen für selbstverwaltetes Apache Kafka erstellen.

Auf dieser Seite wird davon ausgegangen, dass Sie Ihren Kafka-Cluster und das Netzwerk, in dem er sich befindet, bereits ordnungsgemäß konfiguriert haben. Wenn Sie Ihren Cluster oder Ihr Netzwerk einrichten müssen, finden Sie entsprechende Informationen unter the section called “Einrichtung von Cluster und Netzwerk”.

Themen

- Verwenden eines selbstverwalteten Apache-Kafka-Clusters als Ereignisquelle
- Konfiguration der Cluster-Authentifizierungsmethoden in Lambda
- Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine selbstverwaltete Apache-Kafka-Ereignisquelle
- Alle Konfigurationsparameter für selbstverwaltete Apache-Kafka-Ereignisquellen in Lambda

Verwenden eines selbstverwalteten Apache-Kafka-Clusters als Ereignisquelle

Wenn Sie Ihren Apache Kafka- oder Amazon MSK-Cluster als Trigger für Ihre Lambda-Event-Funktion hinzufügen, wird der Cluster als Quelle des Ereignisses.

Lambda liest Ereignisdaten aus den Kafka-Themen, die Sie als angeben `Topics` in einem CreateEventSourceMapping Anfrage, basierend auf Startposition die Sie angeben. Nach erfolgreicher Verarbeitung wird Ihr Kafka-Thema Ihrem Kafka-Cluster zugeordnet.

Lambda liest Nachrichten sequentiell für jede Kafka-Themenpartition. Eine einzelne Lambda-Nutzlast kann Nachrichten von mehreren Partitionen enthalten. Wenn mehr Datensätze verfügbar sind, setzt Lambda die Verarbeitung der Datensätze stapelweise fort, basierend auf BatchSize Wert, den Sie in einem angeben CreateEventSourceMapping Anfrage, bis Ihre Funktion das Thema eingeholt hat.

Nachdem Lambda jeden Batch verarbeitet hat, werden die Offsets der Nachrichten in diesem Batch festgeschrieben. Wenn Ihre Funktion einen Fehler für eine der Nachrichten in einem Batch zurückgibt, wiederholt Lambda den gesamten Nachrichtenbatch, bis die Verarbeitung erfolgreich ist oder die Nachrichten ablaufen. Sie können Datensätze, bei denen alle Wiederholungsversuche fehlschlagen, zur späteren Verarbeitung an ein Ausfallziel senden.

> **Note**
>
> Während Lambda-Ereignisfunktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen die Zuordnungen der Ereignisquellen für Amazon MSK, das selbstverwaltete Apache Kafka, Amazon DocumentDB und Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einer maximalen Zeitüberschreitung von 14 Minuten.

Konfiguration der Cluster-Authentifizierungsmethoden in Lambda

Lambda unterstützt mehrere Methoden zur Authentifizierung mit Ihrem selbstverwalteten Apache-Kafka-Cluster. Stellen Sie sicher, dass Sie den Kafka-Cluster für die Verwendung einer der folgenden unterstützten Authentifizierungsmethoden konfigurieren. Weitere Informationen zur Sicherheit von Kafka finden Sie unter Sicherheit in der Kafka-Dokumentation.

SASL/SCRAM authentication

Lambda unterstützt Simple Authentication and Security Layer/Salted Challenge-Response-Authentifizierungsmechanismus (SASL/SCRAM) Authentifizierung mit Transport Layer Security (TLS) -Verschlüsselung (`SASL_SSL`). Lambda sendet die verschlüsselten Anmeldeinformationen zur Authentifizierung mit dem Cluster. Lambda unterstützt nicht SASL/SCRAM mit Klartext (`SASL_PLAINTEXT`). Für weitere Informationen über SASL/SCRAM Authentifizierung finden Sie unter RFC 5802.

Lambda unterstützt auch SASL/PLAIN Authentifizierung. Da dieser Mechanismus Anmeldeinformationen im Klartext verwendet, muss die Verbindung zum Server TLS-Verschlüsselung verwenden, um sicherzustellen, dass die Anmeldeinformationen geschützt sind.

Für die SASL-Authentifizierung speichern Sie die Anmeldeinformationen als Secret in AWS Secrets Manager. Weitere Informationen zur Verwendung von Secrets Manager finden Sie unter Erstellen Sie eine AWS Secrets Manager Geheimnis in der AWS Secrets Manager Benutzerleitfaden.

> **Important**
>
> Um Secrets Manager für die Authentifizierung zu verwenden, müssen Geheimnisse im selben gespeichert werden AWS Region als Ihre Lambda-Event-Funktion.

Gegenseitige TLS-Authentifizierung

Gegenseitige TLS (mTLS) bietet eine bidirektionale Authentifizierung zwischen Client und Server. Der Client sendet ein Zertifikat an den Server, damit der Server den Client überprüfen kann, und der Server sendet ein Zertifikat an den Client, damit der Client den Server überprüfen kann.

Im selbstverwalteten Apache Kafka fungiert Lambda als Client. Sie konfigurieren ein Client-Zertifikat (als Secret in Secrets Manager), um Lambda für Ihre Kafka-Broker zu authentifizieren. Das Client-Zertifikat muss von einer Zertifizierungsstelle im Trust Store des Servers signiert sein.

Der Kafka-Cluster sendet ein Serverzertifikat an Lambda, um die Kafka-Broker für Lambda zu authentifizieren. Das Serverzertifikat kann ein öffentliches oder ein privates CA-Zertifikat sein CA/self-signed Zertifikat. Das öffentliche CA-Zertifikat muss von einer Zertifizierungsstelle (CA) signiert sein, die sich im Trust Store von Lambda befindet. Für ein privates CA/self-signed Zertifikat, Sie konfigurieren das Server-Root-CA-Zertifikat (als geheimes Zertifikat in Secrets Manager). Lambda verwendet das Root-Zertifikat, um die Kafka-Broker zu verifizieren.

Weitere Informationen über mTLS finden Sie unter Vorstellung von gegenseitiger TLS-Authentifizierung für Amazon MSK als Ereignisquelle.

OAuth-2.0-Authentifizierung

Lambda unterstützt SASL/OAUTHBEARER Authentifizierung mit TLS-Verschlüsselung (`SASL_SSL`). Lambda fordert ein Zugriffstoken von Ihrem OAuth 2.0-Identitätsanbieter an und übergibt dieses Token Ihren Kafka-Brokern. Lambda aktualisiert das Token, bevor es abläuft. Konfigurieren Sie Ihre Broker so, dass sie Token desselben Identitätsanbieters validieren.

Um diese Methode zu verwenden, wählen Sie `OAUTHBEARER_AUTH` als `Type` von einem SourceAccessConfiguration und geben Sie den Secrets Manager-ARN Ihres OAuth-Geheimnisses im `URI` Feld. Den Inhalt des Geheimnisses finden Sie unter the section called “Konfiguration des OAuth-Geheimnisses”.

Lambda unterstützt zwei Arten von Zuschüssen. Die Felder, die Sie im Secret speichern, bestimmen, welches Lambda verwendet.

- Anmeldeinformationen des Kunden — Lambda sendet eine Client-ID und ein Client-Geheimnis an den Token-Endpunkt. Verwenden Sie diesen Gewährungstyp, wenn Ihr Identitätsanbieter ein geheimes Client-Geheimnis ausstellt.
- JWT-Träger — Lambda signiert eine JSON Web Token (JWT) -Assertion mit einem privaten Schlüssel und sendet die Assertion an den Token-Endpunkt. Verwenden Sie diesen Gewährungstyp, wenn Ihr Identitätsanbieter eine signierte Assertion anstelle eines geheimen Client-Schlüssels erwartet.

Wenn Ihr Identitätsanbieter oder Ihre Broker zusätzliche Werte benötigen, geben Sie für jeden Wert einen eigenen Konfigurationseintrag für den Quellzugriff an. Geben Sie den Literalwert in das `URI` Feld statt eines geheimen ARN.

- `OAUTHBEARER_SCOPE` — Der Bereich, den Lambda anfordert, wenn es nach einem Token fragt.
- `OAUTHBEARER_AUDIENCE` — Die Zielgruppe, die Lambda anfordert, wenn es nach einem Token fragt.
- `OAUTHBEARER_LOGICAL_CLUSTER` — Die logische Cluster-ID, die Lambda als SASL-Erweiterung an Ihre Broker sendet.
- `OAUTHBEARER_IDENTITY_POOL` — Die Identitätspool-ID, die Lambda als SASL-Erweiterung an Ihre Broker sendet.

> **Note**
>
> Für diese vier Typen ist ein OAuth 2.0-Authentifizierungstyp erforderlich. Stellen Sie ihnen Folgendes zur Verfügung `OAUTHBEARER_AUTH`, oder nur bereitstellen `OAUTHBEARER_AUDIENCE` mit `IAM_OAUTHBEARER_AUTH`. Wenn Sie einen von ihnen ohne einen dieser Authentifizierungstypen angeben, gibt Lambda einen Validierungsfehler zurück.

Wenn Ihre Broker ein von einer privaten Zertifizierungsstelle signiertes Zertifikat vorlegen, geben Sie außerdem Folgendes an `SERVER_ROOT_CA_CERTIFICATE` damit Lambda sie verifizieren kann. Weitere Informationen finden Sie unter the section called “Konfigurieren des Secrets des Server-Root-CA-Zertifikats”.

IAM-Authentifizierung

Wenn es sich bei Ihrem Cluster um einen Amazon MSK-Cluster handelt, den Sie über Ihr eigenes Netzwerk-Setup erreichen, können Sie ihn als selbstverwaltete Kafka-Ereignisquelle registrieren und sich bei IAM authentifizieren. Zu den benutzerdefinierten Netzwerk-Setups gehört ein benutzerdefinierter Domainname vor Ihren Brokern oder eine Verbindung von einem anderen Konto aus. Lambda signiert jede Verbindung mit den Anmeldeinformationen der Ausführungsrolle Ihrer Funktion, sodass Sie keine Anmeldeinformationen in Secrets Manager speichern.

Um diese Methode zu verwenden, aktivieren Sie die IAM-Zugriffskontrolle auf Ihrem Cluster. Wählen Sie dann `IAM_AUTH` als der `Type` einer Quellzugriffskonfiguration und lassen Sie das weg `URI` Feld. Für die Ausführungsrolle Ihrer Funktion sind die Cluster-Berechtigungen erforderlich, die unter beschrieben sind the section called “Konfigurieren von -Berechtigungen”.

Wenn Ihre Broker ein von einer privaten Zertifizierungsstelle signiertes Zertifikat vorlegen, geben Sie auch Folgendes an `SERVER_ROOT_CA_CERTIFICATE` damit Lambda sie verifizieren kann. Ein benutzerdefinierter Domainname vor Ihren Brokern benötigt dies normalerweise.

IAM-Authentifizierung mit SASL/OAUTHBEARER

Wenn Ihre Broker OAuth 2.0-Token akzeptieren, können Sie Folgendes verwenden AWS als Identitätsanbieter. Lambda präsentiert eine AWS Web-Identitätstoken für Ihre Broker über SASL/OAUTHBEARER. Lambda fordert ein kurzlebiges Token für die Ausführungsrolle Ihrer Funktion mit dem an `sts:GetWebIdentityToken` Aktion, sodass Sie keine Anmeldeinformationen in Secrets Manager speichern. Lambda fordert ein neues Token an, bevor das aktuelle abläuft.

Um diese Methode zu verwenden, wählen Sie `IAM_OAUTHBEARER_AUTH` als `Type` einer Quellzugriffskonfiguration und lassen Sie das weg `URI` Feld. Sie müssen auch angeben `OAUTHBEARER_AUDIENCE` mit dem Publikum, das Ihre Makler erwarten. Die Ausführungsrolle Ihrer Funktion benötigt die Erlaubnis zum Aufrufen `sts:GetWebIdentityToken`.

Konfigurieren Sie Ihre Broker so, dass sie vertrauen AWS als OpenID Connect (OIDC) - Identitätsanbieter, bevor Sie die Ereignisquellenzuordnung erstellen. Wenn Ihre Makler dem nicht vertrauen AWS Emittent, er lehnt das Token ab und Lambda kann nicht aus Ihren Themen lesen.

> **Note**
>
> `IAM_OAUTHBEARER_AUTH` unterstützt nur `OAUTHBEARER_AUDIENCE`. Wenn Sie auch angeben `OAUTHBEARER_SCOPE`, `OAUTHBEARER_LOGICAL_CLUSTER`, oder `OAUTHBEARER_IDENTITY_POOL`, Lambda gibt einen Validierungsfehler zurück.

Konfigurieren des Client-Zertifikat-Secrets

Das Secret CLIENT_CERTIFICATE_TLS_AUTH erfordert ein Zertifikatfeld und ein Feld für einen privaten Schlüssel. Für einen verschlüsselten privaten Schlüssel erfordert das Secret ein Passwort für den privaten Schlüssel. Sowohl das Zertifikat als auch der private Schlüssel müssen im PEM-Format vorliegen.

> **Note**
>
> Lambda unterstützt die PBES1-Algorithmen (aber nicht PBES2) zur Verschlüsselung von privaten Schlüsseln.

Das Zertifikatfeld muss eine Liste von Zertifikaten enthalten, beginnend mit dem Client-Zertifikat, gefolgt von etwaigen Zwischenzertifikaten und endend mit dem Root-Zertifikat. Jedes Zertifikat muss in einer neuen Zeile mit der folgenden Struktur beginnen:

```
-----BEGIN CERTIFICATE-----
            <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager unterstützt Secrets von bis zu 65 536 Bytes, was genügend Platz für lange Zertifikatsketten bietet.

Der private Schlüssel muss im Format PKCS #8 mit folgender Struktur vorliegen:

```
-----BEGIN PRIVATE KEY-----
             <private key contents>
-----END PRIVATE KEY-----
```

Verwenden Sie für einen verschlüsselten privaten Schlüssel die folgende Struktur:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
```

```
               <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

Im folgenden Beispiel sehen Sie den Inhalt eines Secrets für mTLS-Authentifizierung mit einem verschlüsselten privaten Schlüssel. Fügen Sie für einen verschlüsselten privaten Schlüssel das Passwort für den privaten Schlüssel in das Secret ein.

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Konfigurieren des Secrets des Server-Root-CA-Zertifikats

Sie erstellen dieses Secret, wenn Ihre Kafka-Broker TLS-Verschlüsselung mit Zertifikaten verwenden, die von einer privaten CA signiert wurden. Sie können die TLS-Verschlüsselung für VPC verwenden, SASL/SCRAM, SASL/PLAIN, oder mTLS-Authentifizierung.

Das Secret des Server-Root-CA-Zertifikats erfordert ein Feld, das das Root-CA-Zertifikat des Kafka-Brokers im PEM-Format enthält. Das folgende Beispiel zeigt die Struktur des Secrets.

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
```

```
-----END CERTIFICATE-----"
}
```

Konfiguration des OAuth-Geheimnisses

Sie erstellen dieses Geheimnis, wenn Sie es verwenden `OAUTHBEARER_AUTH`. Das Geheimnis ist ein JSON-Dokument. Jedes OAuth-Geheimnis benötigt `oauthTokenEndpointUrl` und `oauthClientId`. Fügen Sie die Felder für nur einen Zuschusstyp hinzu. Wenn Sie Felder aus beiden Zuschusstypen angeben, gibt Lambda einen Validierungsfehler zurück.

Feld	Erforderlich	Description
`oauthTokenEndpointUrl`	Y	Der Token-Endpunkt Ihres Identitätsanbieters. Dies muss eine HTTPS-URL sein.
`oauthClientId`	Y	Die Client-ID, die Ihr Identität sanbieter ausgegeben hat.
`oauthClientSecret`	Nur Kundenanmeldedaten	Das Client-Geheimnis, das Ihr Identitätsanbieter ausgegeben hat.
`oauthClientEmail`	Nur JWT-Inhaber	Die Identität des Dienstkontos, die Lambda in die JWT-Asser tion einfügt.
`oauthPrivateKey`	Nur JWT-Inhaber	Der private Schlüssel, den Lambda zum Signieren der JWT-Assertion verwendet, im PEM-Format.
`oauthTokenExpirationSeconds`	N	Die Lebensdauer der JWT-Assertion, die Lambda signiert. Gilt nur für den JWT-Bearer-Grant-Typ.
`oauthIdpCaCertificate`	N	Das Root-CA-Zertifikat Ihres Identitätsanbieters im PEM-

Feld	Erforderlich	Description
		Format. Geben Sie dies an, wenn Ihr Identitätsanbieter ein von einer privaten CA signiertes Zertifikat vorlegt.

Das folgende Beispiel zeigt ein Geheimnis für den Gewährungstyp „Client-Anmeldeinformationen“.

```
{"oauthTokenEndpointUrl":"https://idp.example.com/oauth2/token",
"oauthClientId":"my-client-id",
"oauthClientSecret":"my-client-secret"}
```

Das folgende Beispiel zeigt ein Geheimnis für den JWT-Bearer-Grant-Typ.

```
{"oauthTokenEndpointUrl":"https://idp.example.com/oauth2/token",
"oauthClientId":"my-client-id",
"oauthClientEmail":"my-service-account@example.com",
"oauthPrivateKey":"-----BEGIN PRIVATE KEY-----
<private key contents>
-----END PRIVATE KEY-----"}
```

> **Important**
>
> Speichern Sie das Geheimnis in demselben AWS Region als Ihre Lambda-Funktion. Lambda liest das Geheimnis mit der Ausführungsrolle Ihrer Funktion, sodass diese Rolle Folgendes benötigt `secretsmanager:GetSecretValue` Erlaubnis für das Geheimnis.

Sie können die Anmeldeinformationen im Secret rotieren, ohne die Zuordnung der Ereignisquelle neu erstellen zu müssen. Lambda nimmt die neuen Werte auf und fordert ein neues Token an.

Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine selbstverwaltete Apache-Kafka-Ereignisquelle

Zum Erstellen einer Zuordnung von Ereignisquellen können Sie die Lambda-Konsole, die AWS Command Line Interface (CLI) oder ein AWS SDK verwenden.

Die folgenden Konsolenschritte fügen einen selbstverwalteten Apache Kafka-Cluster als Auslöser für Ihre Lambda-Ereignisfunktion hinzu. Im Hintergrund wird dadurch eine Zuordnung von Ereignisquellen erstellt.

Voraussetzungen

- Selbstverwaltetes Apache-Kafka-Cluster. Lambda unterstützt Apache Kafka Version 0.10.1.0 und höher.
- Ein Rolle bei der Ausführung mit der Erlaubnis, auf das zuzugreifen AWS Ressourcen, die Ihr selbstverwalteter Kafka-Cluster verwendet.

Hinzufügen eines selbstverwalteten Kafka-Clusters (Konsole)

Folgen Sie diesen Schritten, um Ihren selbstverwalteten Apache Kafka-Cluster und ein Kafka-Thema als Auslöser für Ihre Lambda-Event-Funktion hinzuzufügen.

So fügen Sie Ihrer Lambda-Event-Funktion (Konsole) einen Apache Kafka-Trigger hinzu

1. Öffnen Sie das Seite „Event-Funktionen“ der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Lambda-Event-Funktion.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Führen Sie unter Auslöser-Konfiguration die folgenden Schritte aus:

    a. Wählen Sie den Apache-Kafka-Auslösertyp.

    b. Geben Sie für Bootstrap-Server die Host- und Portpaaradresse eines Kafka-Brokers in Ihrem Cluster ein und wählen Sie dann Hinzufügen. Wiederholen Sie den Vorgang für jeden Kafka-Broker im Cluster.

    c. Geben Sie für Themenname den Namen des Kafka-Themas ein, das zum Speichern von Datensätzen im Cluster verwendet wird.

    d. Wenn Sie den bereitgestellten Modus konfigurieren, geben Sie einen Wert für ein Mindestanzahl an Event-Pollern, ein Wert für Maximale Anzahl von Event-Pollernund ein optionaler Wert für PollerGroupName um die Gruppierung mehrerer ESMs innerhalb derselben Eventquellen-VPC anzugeben.

    e. (Optional) Geben Sie für Batchgröße die maximale Anzahl von Datensätzen ein, die in einem einzelnen Batch empfangen werden sollen.

f. Geben Sie für Batch window (Batch-Fenster) die maximale Zeit in Sekunden ein, die Lambda mit dem Sammeln von Datensätzen verbringt, bevor die Funktion aufgerufen wird.

g. (Optional) Geben Sie für Konsumentengruppen-ID die ID einer Kafka-Konsumentengruppe ein, der Sie beitreten möchten.

h. (Optional) Wählen Sie für Startposition die Option Neueste, um mit dem Lesen des Streams aus dem letzten Datensatz zu beginnen, Horizont trimmen, um mit dem frühesten verfügbaren Datensatz zu beginnen, oder Am Zeitstempel, um einen Zeitstempel anzugeben, ab dem mit dem Lesen begonnen werden soll.

i. (Optional) Wählen Sie bei VPC die Amazon VPC für Ihren Kafka-Cluster aus. Wählen Sie dann VPC subnets (VPC-Subnetze) und VPC security groups (VPC-Sicherheitsgruppen) aus.

   Diese Einstellung ist erforderlich, wenn nur Benutzer innerhalb Ihrer VPC auf Ihre Broker zugreifen.

j. (Optional) Wählen Sie bei Authentication (Authentifizierung) die Option Add (Hinzufügen) aus und gehen Sie dann folgendermaßen vor:

   i. Wählen Sie das Zugriffs- oder Authentifizierungsprotokoll der Kafka-Broker in Ihrem Cluster aus.

      - Wenn Ihr Kafka-Broker verwendet SASL/PLAIN Authentifizierung, wählen Sie BASIC_AUTH.
      - Wenn Ihr Broker verwendet SASL/SCRAM Authentifizierung, wählen Sie eine der SASL_SCRAM protokolle.
      - Falls Sie die mTLS-Authentifizierung konfigurieren, wählen Sie das Protokoll CLIENT_CERTIFICATE_TLS_AUTH aus.
      - Wenn Ihr Broker OAuth 2.0 verwendet, wählen Sie OAUTHBEARER_AUTH.
      - Wenn Ihr Broker die IAM-Zugriffskontrolle verwendet, wählen Sie IAM_AUTH.
      - Wenn Ihr Makler akzeptiert AWS Web-Identitätstoken statt OAuth 2.0, wählen Sie IAM_OAUTHBEARER_AUTH.

   ii. Für SASL/SCRAM, mTLS- oder OAuth 2.0-Authentifizierung, wählen Sie den Secrets Manager-Schlüssel, der die Anmeldeinformationen für Ihren Kafka-Cluster enthält. IAM und IAM mit SASL/OAUTHBEARER Bei der Authentifizierung wird kein Geheimnis verwendet.

k. (Optional) Wählen Sie bei Encryption (Verschlüsselung) das Secrets-Manager-Secret aus, das das Root-CA-Zertifikat enthält, das Ihre Kafka-Broker zur TLS-Verschlüsselung verwenden, falls Ihre Kafka-Broker von einer privaten Zertifizierungsstelle signierte Zertifikate verwenden.

   Diese Einstellung gilt für die TLS-Verschlüsselung für SASL/SCRAM oder SASL/PLAIN, und zur mTLS-Authentifizierung.

l. Um den Auslöser zu Testzwecken in einem deaktivierten Zustand zu erstellen (empfohlen), deaktivieren Sie Auslöser aktivieren. Um den Auslöser sofort zu aktivieren, wählen Sie Auslöser aktivieren.

5. Wählen Sie hinzufügen aus, um den Auslöser zu erstellen.

Hinzufügen eines selbstverwalteten Kafka-Clusters (AWS CLI)

Verwenden Sie das folgende Beispiel AWS CLI Befehle zum Erstellen und Anzeigen eines selbstverwalteten Apache Kafka-Triggers für Ihre Lambda-Ereignisfunktion.

Verwenden SASL/SCRAM

Wenn Kafka-Benutzer über das Internet auf Ihre Kafka-Broker zugreifen, geben Sie das Secrets Manager-Geheimnis an, für das Sie erstellt haben SASL/SCRAM Authentifizierung. Das folgende Beispiel verwendet create-event-source-mapping AWS CLI Befehl zum Zuordnen einer Lambda-Event-Funktion mit dem Namen `my-kafka-function` zu einem Kafka-Thema mit dem Namen `AWSKafkaTopic`.

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

Verwenden einer VPC

Wenn nur Kafka-Benutzer in Ihrer VPC auf Ihre Kafka-Broker zugreifen, müssen Sie Ihre VPC, Subnetze und VPC-Sicherheitsgruppe angeben. Das folgende Beispiel verwendet create-event-source-mapping AWS CLI Befehl zum Zuordnen einer Lambda-Event-Funktion mit dem Namen `my-kafka-function` zu einem Kafka-Thema mit dem Namen `AWSKafkaTopic`.

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

Den Status mit dem anzeigen AWS CLI

Im folgenden Beispiel wird das verwendet get-event-source-mapping AWS CLI Befehl zur Beschreibung des Status der Ereignisquellenzuordnung, die Sie erstellt haben.

```
aws lambda get-event-source-mapping
             --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

Alle Konfigurationsparameter für selbstverwaltete Apache-Kafka-Ereignisquellen in Lambda

Alle Lambda-Ereignisquelltypen haben dasselbe CreateEventSourceMapping und UpdateEventSourceMapping API-Operationen. Allerdings gelten nur einige der Parameter für selbstverwaltetes Apache Kafka, wie in der folgenden Tabelle dargestellt.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
SelfManagedKafkaEventSourceConfig.ConsumptionMode	N	Streamen	`Stream` oder `Queue`. Legt fest, wie Lambda Datensätze aus dem Topic verwendet. Kann nur bei Create gesetzt werden. Wann `Queue`, `StartingPosition` , `StartingPositionTimestamp` , und

Parameter	Erforderlich	Standard	Hinweise
			`MaximumBatchingWindowInSeconds` nicht anwenden. Details hierzu finden Sie unter Abfragen von Kafka-Themen im Warteschlangenmodus.
DestinationConfig	N	N/A	the section called “Fehlgeschlagene Aufrufe beibehalten”
Aktiviert	N	Wahr	
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet
FunctionName	Y	N/A	
KMSKeyArn	N	N/A	the section called “Verschlüsselung der Filterkriterien”
MaximumBatchingWindowInSeconds	N	500 ms	Batching-Verhalten Gilt nicht, wenn `ConsumptionMode` ist `Queue`.

Parameter	Erforderlich	Standard	Hinweise
ProvisionedPollers Config	N	`MinimumPollers` : Standardwert von 1, wenn nicht angegeben `MaximumPollers` : Standardwert von 200, wenn nicht angegeben `PollerGroupName` : N/A	the section called “Modus bereitgestellter Kapazität”
SelfManagedEventSource	Y	N/A	Liste der Kafka Broker. Kann nur auf „Erstellen“ festgelegt werden
SelfManagedKafkaEventSourceConfig	N	Enthält die ConsumerGroupId Feld, das standardmäßig einen eindeutigen Wert hat.	Kann nur auf „Erstellen“ festgelegt werden
SourceAccessConfigurations	N	Keine Anmeldeinformationen	VPC-Informationen oder Authentifizierungsanmeldeinformationen für den Cluster Für SASL_PLAIN auf BASIC_AUTH setzen

Parameter	Erforderlich	Standard	Hinweise
StartingPosition	Y (Nur Stream-Modus)	N/A	AT_TIMESTAMP, TRIM_HORIZON, oder LATEST Kann nur auf „Erstellen“ festgelegt werden Gilt nicht wenn `ConsumptionMode` ist `Queue`.
StartingPositionTimestamp	N	N/A	Erforderlich wenn StartingPosition ist auf AT_TIMESTAMP gesetzt Gilt nicht wenn `ConsumptionMode` ist `Queue`.
Tags (Markierungen)	N	N/A	the section called “Tags für die Zuordnung von Ereignisquellen”
Topics	Y	N/A	Topic-Name Kann nur auf „Erstellen“ festgelegt werden

> **Note**
>
> Wenn Sie eine angeben `PollerGroupName`, mehrere ESMs innerhalb derselben Amazon VPC können die Kapazität der Event Poller Unit (EPU) gemeinsam nutzen. Sie können diese Option verwenden, um die Kosten für Ihre ESMs im Bereitstellungsmodus zu optimieren. Anforderungen für die ESM-Gruppierung:

- ESMs müssen sich in derselben Amazon VPC befinden
- Maximal 100 ESMs pro Umfragegruppe
- Die aggregierte maximale Anzahl von Pollern für alle ESMs in einer Gruppe darf 2000 nicht überschreiten

Sie können das aktualisieren `PollerGroupName` um einen ESM in eine andere Gruppe zu verschieben oder einen ESM aus einer Gruppe zu entfernen, indem Sie Folgendes festlegen `PollerGroupName` zu einer leeren Zeichenfolge („“).

Skalierungsmodi des Ereignis-Pollers von Apache Kafka in Lambda

Sie können zwischen zwei Skalierungsmodi von Ereignis-Pollern für Zuordnungen von Ereignisquellen von Amazon MSK und selbstverwaltetem Apache Kafka wählen.

- On-demand Modus (Standard)
- Modus bereitgestellter Kapazität

On-demand Modus (Standard)

Wenn Sie zuerst eine Kafka-Ereignisquelle erstellen, weist Lambda eine Standardanzahl von Ereignis-Pollern zu, um alle Partitionen im Kafka-Thema zu verarbeiten. Lambda erhöht oder verringert automatisch die Anzahl der Ereignis-Poller je nach Nachrichtenlast.

In Intervallen von einer Minute wertet Lambda den Offset-Lag aller Partitionen des Themas aus. Wenn die Offset-Verzögerung zu hoch ist, empfängt die Partition Nachrichten schneller als Lambda sie verarbeiten kann. Falls erforderlich, fügt Lambda dem Thema Ereignis-Poller hinzu oder entfernt sie. Dieser Prozess der automatischen Skalierung durch Hinzufügen oder Entfernen von Ereignis-Pollern erfolgt innerhalb von drei Minuten nach der Auswertung.

Wenn Ihre Lambda-Zielfunktion gedrosselt ist, reduziert Lambda die Anzahl der Event-Poller. Diese Aktion verringert den Workload der Funktion, indem sie die Anzahl der Nachrichten reduziert, die Ereignis-Poller abrufen und an die Funktion senden können.

Modus bereitgestellter Kapazität

Für Workloads, bei denen Sie den Durchsatz Ihrer Zuordnung von Ereignisquellen optimieren müssen, können Sie den Bereitstellungsmodus verwenden. Im Bereitstellungsmodus definieren Sie Mindest- und Höchstgrenzen für die Anzahl der bereitgestellten Ereignis-Poller. Diese bereitgestellten Event-Poller sind auf Ihre Zuordnung von Ereignisquellen ausgerichtet und können unerwartete Nachrichtenspitzen durch reaktionsschnelle Autoskalierung bewältigen. Wir empfehlen die Verwendung des Bereitstellungsmodus für Kafka-Workloads, die strenge Leistungsanforderungen haben.

In Lambda ist ein Event Poller eine Recheneinheit mit Durchsatzmöglichkeiten, die je nach Art der Ereignisquelle variieren. Bei Amazon MSK und selbstverwaltetem Apache Kafka kann jeder Event Poller bis zu 5% MB/sec des Durchsatzes oder bis zu 5 gleichzeitige Aufrufe verarbeiten. Wenn Ihre Ereignisquelle beispielsweise eine durchschnittliche Nutzlast von 1 MB erzeugt und die

durchschnittliche Dauer Ihrer Funktion 1 Sekunde beträgt, kann ein einziger Kafka-Event-Poller 5 MB/sec Durchsatz- und 5 gleichzeitige Lambda-Aufrufe unterstützen (vorausgesetzt, es gibt keine Payload-Transformation). Für Amazon SQS kann jeder Event Poller bis zu 1 MB/sec Durchsatz oder bis zu 10 gleichzeitige Aufrufe verarbeiten. Bei der Verwendung des Bereitstellungsmodus fallen zusätzliche Kosten an, die von Ihrer Nutzung des Event-Pollers abhängen. Details zu den Preisen finden Sie unter AWS Lambda -Preise.

> **Note**
>
> Wenn Sie den Bereitstellungsmodus verwenden, müssen Sie im Rahmen Ihrer Netzwerkkonfiguration keine AWS PrivateLink VPC-Endpoints erstellen oder die zugehörigen Berechtigungen gewähren.

Im Bereitstellungsmodus liegt der Bereich der akzeptierten Werte für die Mindestanzahl der Ereignis-Poller (`MinimumPollers`) zwischen 1 und 200 (einschließlich). Der Bereich der akzeptierten Werte für die maximale Anzahl von Event-Pollern (`MaximumPollers`) liegt zwischen 1 und einschließlich 2.000. `MaximumPollers` muss größer als oder gleich `MinimumPollers` sein. Um eine geordnete Verarbeitung innerhalb der Partitionen zu gewährleisten, begrenzt Lambda das `MaximumPollers` auf die Anzahl der Partitionen im Thema.

Weitere Einzelheiten zur Auswahl geeigneter Werte für minimale und maximale Ereignis-Poller finden Sie unter Bewährte Methoden.

Sie können den Bereitstellungsmodus für Ihre Kafka-Zuordnung von Ereignisquellen über die Konsole oder die Lambda-API konfigurieren.

Konfigurieren des Bereitstellungsmodus für eine vorhandene Zuordnung von Ereignisquellen (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion mit der Zuordnung von Ereignisquellen aus, für die Sie den Bereitstellungsmodus konfigurieren möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie die Zuordnung von Ereignisquellen, für die Sie den Bereitstellungsmodus konfigurieren möchten, und wählen Sie dann Bearbeiten.
5. Wählen Sie unter Bereitstellungsmodus die Option Konfigurieren aus.

- Geben Sie für Minimum Event Pollers einen Wert zwischen 1 und 200 ein. Wenn Sie keinen Wert angeben, wählt Lambda einen Standardwert von 1.
- Geben Sie für Maximum Event Pollers einen Wert zwischen 1 und 2000 ein. Dieser Wert muss größer oder gleich Ihrem Wert für Minimum Event Pollers sein. Wenn Sie keinen Wert angeben, wählt Lambda einen Standardwert von 200.

6. Wählen Sie Speichern.

Sie können den Bereitstellungsmodus programmgesteuert mithilfe des ProvisionedPollerConfigObjekts in Ihrem konfigurieren. EventSourceMappingConfiguration Der folgende UpdateEventSourceMappingCLI-Befehl konfiguriert beispielsweise einen `MinimumPollers` Wert von 5 und einen `MaximumPollers` Wert von 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

Nachdem Sie den Bereitstellungsmodus konfiguriert haben, können Sie die Verwendung von Event-Pollern für Ihren Workload beobachten, indem Sie die `ProvisionedPollers`-Metrik überwachen. Weitere Informationen finden Sie unter the section called “Metriken zur Zuordnung von Ereignisquellen”.

Um den Bereitstellungsmodus zu deaktivieren und zum Standardmodus (auf Abruf) zurückzukehren, können Sie den folgenden UpdateEventSourceMappingCLI-Befehl verwenden:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

Erweiterte Funktionen zur Fehlerbehandlung und Leistung

Für Kafka-Ereignisquellenzuordnungen mit aktiviertem Bereitstellungsmodus können Sie zusätzliche Funktionen konfigurieren, um die Fehlerbehandlung und Leistung zu verbessern:

- Wiederholungskonfigurationen — Steuern Sie, wie Lambda mit fehlgeschlagenen Datensätzen umgeht, indem Sie maximale Wiederholungsversuche, Altersbeschränkungen für Datensätze, Batch-Splitting und partielle Batch-Antworten festlegen.

- Kafka-Ziele bei einem Ausfall — Senden Sie fehlgeschlagene Datensätze zur späteren Verarbeitung oder Analyse an ein Kafka-Thema.

Bewährte Verfahren und Überlegungen bei der Verwendung des Bereitstellungsmodus

Die optimale Konfiguration der minimalen und maximalen Ereignispoller für Ihre Zuordnung von Ereignisquellen hängt von den Leistungsanforderungen Ihrer Anwendung ab. Wir empfehlen Ihnen, mit den standardmäßigen Minimum Event Pollers zu beginnen, um das Leistungsprofil zu erstellen. Passen Sie Ihre Konfiguration auf der Grundlage der beobachteten Nachrichtenverarbeitungsmuster und Ihres gewünschten Leistungsprofils an.

Erhöhen Sie bei Workloads mit starkem Datenverkehr und strengen Leistungsanforderungen die Mindestanzahl der Event-Poller, um plötzliche Nachrichtenfluten zu bewältigen. Um den Mindestbedarf an Event-Pollern zu ermitteln, berücksichtigen Sie die Nachrichten pro Sekunde und die durchschnittliche Größe der Nutzdaten Ihres Workloads und verwenden Sie die Durchsatzkapazität eines einzelnen Event-Pollers (bis zu 5 MBps) als Referenz.

Um eine geordnete Verarbeitung innerhalb einer Partition aufrechtzuerhalten, begrenzt Lambda die maximale Anzahl der Event-Poller auf die Anzahl der Partitionen im Thema. Darüber hinaus hängt die maximale Anzahl von Event-Pollern, auf die Ihr Zuordnung von Ereignisquellen skaliert werden kann, von den Gleichzeitigkeitseinstellungen der Funktion ab.

Wenn Sie den Bereitstellungsmodus aktivieren, aktualisieren Sie Ihre Netzwerkeinstellungen, um AWS PrivateLink VPC-Endpunkte und zugehörige Berechtigungen zu entfernen.

Kostenoptimierung für den Bereitstellungsmodus

Preisgestaltung im Bereitstellungsmodus

Der Bereitstellungsmodus wird auf der Grundlage der bereitgestellten Mindestanzahl an Event-Pollern und der Anzahl der während der automatischen Skalierung verbrauchten Event-Poller berechnet. Die Gebühren werden anhand einer Abrechnungseinheit berechnet, die als Event Poller Unit (EPU) bezeichnet wird. Sie zahlen für die Anzahl und Dauer der verwendeten EPUs, gemessen in. Event-Poller-Unit-hours

Sie können den Bereitstellungsmodus entweder mit einem einzigen ESM für leistungssensitive Anwendungen verwenden oder Sie können mehrere ESMs innerhalb derselben VPC gruppieren, um EPU-Kapazität und Kosten gemeinsam zu nutzen. In den folgenden Abschnitten werden zwei Funktionen beschrieben, mit denen Sie Ihre Kosten für den Bereitstellungsmodus optimieren können. Preisdetails finden Sie unter AWS Lambda-Preise.

Verbesserte EPU-Nutzung

Jede EPU unterstützt bis zu 20 MB/s Durchsatzkapazitäten für Ereignisabfragen und unterstützt standardmäßig 10 Ereignisabfragen. Wenn Sie einen Bereitstellungsmodus für Kafka ESM erstellen, indem Sie Mindest- und Höchstwerte für Poller festlegen, verwendet er die Mindestanzahl an Pollern, um EPUs auf der Grundlage der Standardeinstellung von 10 Event-Pollern pro EPU bereitzustellen. Jeder Event-Poller kann jedoch unabhängig skaliert werden, um bis zu 5% MB/s der Durchsatzkapazität zu unterstützen, was möglicherweise eine geringere Dichte von Event-Pollern auf einer bestimmten EPU erfordert und eine Skalierung von EPUs auslösen kann. Die Anzahl der Event Poller, die einer EPU zugewiesen sind, hängt von der Rechenkapazität ab, die von jedem Event Poller verbraucht wird. Dieser Ansatz der verbesserten EPU-Nutzung ermöglicht es Event-Pollern mit unterschiedlichen Durchsatzanforderungen, die EPU-Kapazität effektiv zu nutzen, wodurch die Kosten für alle ESMs gesenkt werden.

ESM-Gruppierung

Um Ihre Kosten für den Bereitstellungsmodus weiter zu optimieren, können Sie mehrere Kafka-ESMs gruppieren, um die EPU-Kapazität gemeinsam zu nutzen. Mit ESM-Gruppierung und verbesserter EPU-Auslastung können Sie Ihre Kosten für Workloads mit geringem Durchsatz im Bereitstellungsmodus um bis zu 90% senken, verglichen mit der Ausführung im Einzel-ESM-Modus. Alle ESMs, die weniger als 1 EPU-Kapazität benötigen, profitieren von der ESM-Gruppierung. Solche ESMs benötigen in der Regel nur wenige Mindestanzahl an Event-Pollern, um ihren Durchsatzbedarf zu decken. Mit dieser Funktion können Sie den Bereitstellungsmodus für all Ihre Kafka-Workloads verwenden und von Funktionen wie Schemavalidierung, Filterung von Avro/Protobuf Ereignissen, Aufrufen mit geringer Latenz und verbesserter Fehlerbehandlung profitieren, die nur im Bereitstellungsmodus verfügbar sind.

Wenn Sie den `PollerGroupName` Parameter mit demselben Wert für mehrere ESMs innerhalb derselben Amazon VPC konfigurieren, teilen sich diese ESMs EPU-Ressourcen, anstatt jeweils eine eigene EPU-Kapazität zu benötigen. Sie können bis zu 100 ESMs pro Poller-Gruppe gruppieren, und die Gesamtzahl der Poller für alle ESMs in einer Gruppe darf 2000 nicht überschreiten.

Um die ESM-Gruppierung zu konfigurieren (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Ihre Funktion.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wenn Sie eine neue Kafka-Ereignisquellenzuordnung erstellen oder eine bestehende bearbeiten, wählen Sie im Bereitstellungsmodus die Option Konfigurieren aus.

5. Geben Sie für Minimum Event Pollers einen Wert zwischen 1 und 200 ein.
6. Geben Sie für Maximum Event Pollers einen Wert zwischen 1 und 2000 ein.
7. Geben Sie unter Poller-Gruppenname einen Bezeichner für die Gruppe ein. Verwenden Sie denselben Namen für andere ESMs, die Sie gruppieren möchten.
8. Wählen Sie Speichern.

Um die ESM-Gruppierung zu konfigurieren (AWS CLI)

Im folgenden Beispiel wird ein ESM mit einer Poller-Gruppe namens erstellt: `production-app-group`

```
aws lambda create-event-source-mapping \
  --function-name myFunction1 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic1 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

Um derselben Gruppe einen weiteren ESM hinzuzufügen (mit gemeinsamer EPU-Kapazität), verwenden Sie dasselbe: PollerGroupName

```
aws lambda create-event-source-mapping \
  --function-name myFunction2 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic2 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

> **Note**
>
> Sie können den aktualisierenPollerGroupName, um einen ESM in eine andere Gruppe zu verschieben, oder einen ESM aus einer Gruppe entfernen, indem Sie eine leere Zeichenfolge („“) übergeben für: PollerGroupName

```
# Move ESM to a different group
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "new-group-name"
  }'

# Remove ESM from group (use dedicated resources)
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": ""
  }'
```

Überlegungen zur Gruppierungsstrategie

- Anwendungsgrenzen — Gruppieren Sie ESMs, die zu denselben Anwendungen oder Diensten gehören, um die Kostenverteilung und Verwaltung zu verbessern. Erwägen Sie die Verwendung von Namenskonventionen wie app-name-environment (zum Beispielorder-processor-prod).
- Verkehrsmuster — Vermeiden Sie es, ESMs mit hohem Durchsatz und hohem Datenverkehrsmuster zu gruppieren, da dies zu Ressourcenkonflikten führen kann.
- Explosionsradius — Berücksichtigen Sie die Auswirkungen, wenn Probleme mit der gemeinsam genutzten Infrastruktur auftreten. Alle ESMs derselben Gruppe sind von Beschränkungen gemeinsam genutzter Ressourcen betroffen. Für geschäftskritische Workloads möchten Sie möglicherweise separate Gruppen oder dedizierte ESMs verwenden.

Beispiel für Kostenoptimierung

Stellen Sie sich ein Szenario vor, in dem Sie 10 ESMs haben, die jeweils mit einem Event-Poller konfiguriert sind und der Durchsatz unter 2 liegt: MB/s

Ohne Gruppierung:

- Jeder ESM benötigt eine eigene EPU
- Insgesamt benötigte EPUs: 10
- Kosten pro EPU: 0$. 185/hour im Osten der USA (Nord-Virginia)
- Monatliche EPU-Kosten (720 Stunden): 10 × 720 × 0,185$ = 1.332$

Mit Gruppierung:

- Alle 10 ESMs teilen sich die EPU-Kapazität
- 10 Event-Poller passen in 1 EPU (mit neuer Unterstützung für 10 Poller pro EPU)
- Insgesamt benötigte EPUs: 1
- Monatliche EPU-Kosten (720 Stunden): 1 × 720 × 0,185$ = 133,20$
- Kosteneinsparungen: 90% (Einsparungen von 1.198,80 USD pro Monat)

Apache-Kafka-Abfrage und Stream-Startpositionen in Lambda

Der StartingPosition Parameter teilt Lambda mit, wann mit dem Lesen von Nachrichten aus Ihrem Amazon MSK- oder selbstverwalteten Apache Kafka-Stream begonnen werden soll. Es stehen drei Optionen zur Auswahl:

- Neueste – Lambda beginnt unmittelbar nach dem neuesten Datensatz im Kafka-Thema mit dem Lesen.
- Horizont trimmen –Lambda beginnt ab dem letzten ungetrimmten Datensatz im Kafka-Thema mit dem Lesen. Dies ist auch der älteste Datensatz im Thema.
- Am Zeitstempel – Lambda beginnt an einer durch einen Zeitstempel definierten Position in Unix-Zeitsekunden mit dem Lesen. Verwenden Sie den StartingPositionTimestamp Parameter, um den Zeitstempel anzugeben.

Die Stream-Abfrage während der Erstellung oder Aktualisierung einer Zuordnung von Ereignisquellen ist letztendlich konsistent:

- Bei der Erstellung der Ereignisquellenzuordnung kann es mehrere Minuten dauern, bis Ereignisse aus dem Stream abgefragt werden.
- Bei Aktualisierungen der Ereignisquellenzuordnung kann es bis zu 90 Sekunden dauern, bis die Abfrage von Ereignissen aus dem Stream beendet und erneut gestartet wird.

Dieses Verhalten bedeutet, dass die Zuordnung von Ereignisquellen Ereignisse während der Erstellung oder Aktualisierung verpassen könnte, wenn Sie `LATEST` als Startposition für den Stream angeben. Um sicherzustellen, dass keine Ereignisse verpasst werden, geben Sie entweder `TRIM_HORIZON` oder `AT_TIMESTAMP` an.

Abfragen von Kafka-Themen im Warteschlangenmodus

Der Warteschlangenmodus ist ein Konsummodus, der Share-Gruppen verwendet. Dieser Modus wurde in Apache Kafka 4.0 (für den Kafka Queue Poller ist Version 4.2 oder höher erforderlich). Im Warteschlangenmodus können Sie Nachrichten zu Kafka-Themen mithilfe von Share-Gruppen ungeordnet und mit hohem Durchsatz verarbeiten. Im Gegensatz zu herkömmlichen Kafka-Benutzergruppen, die Partitionen ausschließlich Verbrauchern zuweisen, ermöglichen Freigabegruppen mehreren Verbrauchern das gleichzeitige Lesen derselben Partitionen. Dies ähnelt der Verarbeitung von Nachrichten aus einer Warteschlange durch Amazon SQS.

Wenn Sie eine Ereignisquellenzuordnung für eine selbstverwaltete Apache Kafka Cluster, Sie können zwischen zwei Nutzungsmodi wählen:

- Streamen (Standard). Verwendet Kafka-Benutzergruppen mit einer Reihenfolge pro Partition. Lambda weist jede Partition einem Verbraucher zu.
- Warteschlange. Verwendet Share-Gruppen, die jetzt verfügbar sind in Apache Kafka 4.0, für ungeordnete Verarbeitung mit hohem Durchsatz (für den Kafka Queue Poller ist Version 4.2 oder höher erforderlich). Bei Share Groups können mehrere Verbraucher gleichzeitig von denselben Partitionen lesen, ähnlich wie Amazon SQS Nachrichten aus einer Warteschlange verarbeitet.

Wenn Sie keinen Nutzungsmodus angeben, verwendet die Zuordnung der Ereignisquelle Stream. Diese Seite behandelt den Queue-Nutzungsmodus. Allgemeine Informationen zur Verwendung von Lambda mit Selbstverwaltung Apache Kafka, siehe Verwendung von Lambda mit selbstverwaltetem Apache Kafka.

Im Warteschlangenmodus verwendet Lambda den Kafka Share Consumer, um Nachrichten aus dem Thema abzufragen. Jeder Event-Poller verarbeitet Nachrichten über alle Partitionen hinweg ohne Reihenfolge auf Partitionsebene. Dadurch wird die Parallelität für jeden Poller erhöht.

Mindestanforderungen

Um den Queue-Nutzungsmodus verwenden zu können, muss Ihr Kafka-Cluster die folgenden Anforderungen erfüllen:

- Apache Kafka Version 4.2 oder höher mit aktivierten Share-Gruppen. Kafka aktiviert `share.version` standardmäßig, und Sie können es verwalten über `kafka-features.sh`.
- Self-managed Apache Kafka Nur (SMK) -Cluster. Von Amazon verwaltetes Streaming für Apache Kafka (Amazon MSK) unterstützt derzeit den Warteschlangenmodus nicht. AWS fügt MSK-Unterstützung hinzu, sobald MSK Kafka 4.2 oder höher mit aktivierten Share-Gruppen anbietet.

Vergleich von Stream und Queue

Feature	Stream (Standard)	Warteschlange
Bestellung	Per-partition Bestellung garantiert	Keine Bestellgarantie
Modell für Verbraucher	Verbrauchergruppe (exklusive Partitionszuweisung)	Gruppe teilen (gleichzeitiger Partitionszugriff)
Durchsatz pro Event-Poller	Bis zu 5 MB/s	Bis zu 5 MB/s
Gleichzeitige Aufrufe pro Event-Poller	Bis zu 5	Bis zu 10
Modi skalieren	On-demand und bereitgestellt	On-demand und bereitgestellt
Gemeinsame Nutzung von EPUs	Kann EPUs mit anderen Kafka Streams-Zuordnungen von Eventquellen teilen	Isoliert. Quellzuordnungen für Warteschlangenerei gnisse werden nur mit anderen Quellzuordnungen für Warteschlangenereignisse gemeinsam genutzt
Anwendungsfall	Geordnete Verarbeitung von Ereignissen	High-throughput ungeordnete Arbeitslasten

Warteschlangenzuordnung erstellen

Um eine Ereignisquellenzuordnung im Warteschlangenmodus zu erstellen, legen Sie den `ConsumptionMode` Feld von `SelfManagedKafkaEventSourceConfig` zu `Queue`. Ersetzen Sie in den folgenden Beispielen `my-function`, die Broker-Endpunkte und `my-queue-topic` mit deinen eigenen Werten.

AWS CLI

Folgendes AWS Der CLI-Befehl erstellt eine Zuordnung der Ereignisquelle im Warteschlangenmodus. Ersetzen *`my-function`* mit Ihrem Funktionsnamen, den Broker-Endpunkten bei Ihren Brokern und *`my-queue-topic`* mit deinem Thema.

```
PROMPT> aws lambda create-event-source-mapping \
  --function-name my-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":["broker1.us-west-2.example.com:9092","broker2.us-west-2.example.com:9092"]}}' \
  --topics my-queue-topic \
  --batch-size 100 \
  --self-managed-kafka-event-source-config '{"ConsumptionMode": "Queue"}' \
  --provisioned-poller-config '{"MinimumPollers": 1, "MaximumPollers": 10}'
```

AWS CloudFormation

Folgendes AWS CloudFormation Die Vorlage erstellt eine Zuordnung der Ereignisquelle im Warteschlangenmodus. Ersetzen Sie die Broker-Endpunkte und *my-queue-topic* mit deinen eigenen Werten.

```
MyKafkaQueueESM:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    FunctionName: !Ref MyFunction
    SelfManagedEventSource:
      Endpoints:
        KafkaBootstrapServers:
          - broker1.us-west-2.example.com:9092
          - broker2.us-west-2.example.com:9092
    Topics:
      - my-queue-topic
    BatchSize: 100
    SelfManagedKafkaEventSourceConfig:
      ConsumptionMode: Queue
    ProvisionedPollerConfig:
      MinimumPollers: 1
      MaximumPollers: 10
```

Konfigurationsparameter

In der folgenden Tabelle sind die Parameter aufgeführt, die für die Zuordnungen von Ereignisquellen im Warteschlangenmodus gelten.

Parameter	Description	Bereich/Standard
`BatchSize`	Maximale Anzahl von Datensätzen pro Stapel, die an die Funktion gesendet werden	1—10.000. Standard: 100
`Enabled`	Ob die Zuordnung der Ereignisquelle aktiv ist	`true` oder `false`. Standard: `true`
`FilterCriteria`	Muster zur Ereignisfilterung, um nur übereinstimmende Datensätze zu verarbeiten	Bis zu 5 Filtermuster
`MaximumRetryAttempts`	Maximale Anzahl von Wiederholungsversuchen für fehlgeschlagene Datensätze	-1—10.000. Standard: 0 (im Warteschlangenmodus führt der Kafka-Broker Wiederholungsversuche durch, wenn kein Ziel für den Fall eines Fehlers konfiguriert ist)
`BisectBatchOnFunctionError`	Bei einem Funktionsfehler den Stapel in zwei Hälften teilen	`true` oder `false`. Standard: `false`
`ProvisionedPollerConfig.MinimumPollers`	Mindestanzahl an Event-Pollern (bereitgestellter Modus)	1—200. Standard: 1
`ProvisionedPollerConfig.MaximumPollers`	Maximale Anzahl von Event-Pollern (bereitgestellter Modus)	1—2.000. Standard: 200
`ProvisionedPollerConfig.PollerGroupName`	Gruppenname für die gemeinsame Nutzung von EPU (Bereitstellungsmodus, nur Quelle-Zuordnungen für Warteschlangenereignisse)	Zeichenfolge. Bis zu 100 Zuordnungen von Ereignisquellen pro Gruppe

Parameter	Description	Bereich/Standard
`SelfManagedKafkaEventSourceConfig.ConsumptionMode`	Nutzungsmodus für die Ereignisquelle	Muss sein `Queue` für den Warteschlangenmodus
`SchemaRegistryConfig`	Schemaregistrierung für die Ereignisvalidierung	Siehe Verwenden von Schemaregistern mit Kafka-Ereignisquellen in Lambda

> **Important**
>
> Das `MaximumBatchingWindowInSeconds` Parameter ist wird nicht unterstützt für Zuordnungen von Ereignisquellen im Warteschlangenmodus. Der Kafka Share Consumer verwendet eine ausdrückliche Bestätigung. Alle Datensätze in einem Stapel müssen explizit bestätigt werden, bevor der Verbraucher neue Datensätze abrufen kann. Dies ist mit einem zeitbasierten Batching-Fenster nicht kompatibel. Wenn Sie angeben `MaximumBatchingWindowInSeconds` Beim Erstellen einer Ereignisquellenzuordnung im Warteschlangenmodus gibt die API einen Validierungsfehler zurück.

Unterstützte Funktionen zur Zuordnung von Ereignisquellen

In der folgenden Tabelle sind die Funktionen zur Zuordnung von Lambda-Ereignisquellen aufgeführt, die im Warteschlangenmodus verfügbar sind.

Funktionsmerkmal	Unterstützt	Hinweise
Modus bereitgestellter Kapazität	Ja	Konfigurieren `MinimumPollers` und `MaximumPollers`
On-demand Modus	Ja	Standard-Skalierungsmodus. Lambda skaliert Poller automatisch auf bis zu 5 MB/s oder 50 gleichzeitige Aufrufe, je nachdem, welcher Wert

Funktionsmerkmal	Unterstützt	Hinweise
		zuerst erreicht wird. Keine zusätzliche Konfiguration erforderlich.
Ereignisfilterung	Ja	Bis zu 5 Filtermuster pro Zuordnung der Ereignisquelle
Schema Registry	Ja	Avro und Protobuf mit Glue Schema Registry oder Confluent Schema Registry. Die Filterung der Schemareg istrierung ist nur im bereitges tellten Modus verfügbar.
Low-latency Verarbeitung	Ja	Im bereitgestellten Modus verfügbar
Versuchen Sie es erneut mit den Konfigurationen	Ja	`MaximumRetryAttempts` , `BisectBatchOnFunct ionError`
On-failure Ziel (Kafka-Thema)	Ja	Fehlgeschlagene Datensätze an ein Kafka-Thema senden
On-failure Ziel (SNS/SQS)	Ja	Fehlgeschlagene Datensätze an SNS oder SQS senden
Protokollierung der Zuordnung der Ereignisquelle	Ja	CloudWatch Protokolle für die Diagnose der Ereignisq uellenzuordnung

Funktionsmerkmal	Unterstützt	Hinweise
Gruppierung der Zuordnung von Ereignisquellen (`PollerGroupName`)	Ja	Zuordnungen von Ereignisquellen im Warteschlangenmodus können EPUs nur mit anderen Ereignisquellenzuordnungen im Warteschlangenmodus gemeinsam nutzen (bereitgestellter Modus)
Fenster zum Stapeln	Nein	Wird aufgrund des Verhaltens im Share Consumer-Blockiermodus nicht unterstützt
Startposition	Nein	`StartingPosition` und `StartingPositionTimestamp` werden nicht unterstützt. Der Warteschlangenmodus liest immer aus der Commit-Position der Share-Gruppe.

Skalierungsmodi

Die Zuordnungen der Ereignisquellen im Warteschlangenmodus unterstützen sowohl den bedarfsgesteuerten (Standard) als auch den bereitgestellten Skalierungsmodus.

On-demand Modus (Standard)

Im On-Demand-Modus verwaltet Lambda automatisch Event-Poller für die Zuordnung Ihrer Ereignisquellen im Warteschlangenmodus.

On-demand Der Modus skaliert auf bis zu 5 Event-Poller pro Eventquellen-Mapping. Er erreicht seine Obergrenze entweder bei 5 MB/s Gesamtdurchsatz oder 50 gleichzeitige Aufrufe (5 Poller mal 10). Dies geschieht unabhängig davon, welches Limit zuerst erreicht wird. Verwenden Sie den bereitgestellten Modus, um über diese Grenzen hinaus zu skalieren. On-demand Für den Modus

ist neben den standardmäßigen Zuordnungsparametern für die Ereignisquelle keine zusätzliche Konfiguration erforderlich.

In der folgenden Tabelle sind die Grenzwerte für den On-Demand-Modus aufgeführt.

Limit	Wert
Durchsatz pro Event-Poller	Bis zu 1 MB/s
Gleichzeitige Aufrufe pro Event-Poller	Bis zu 10
Maximale Anzahl von Event-Pollern	5
Maximaler Gesamtdurchsatz	Bis zu 5 MB/s (5 Poller mal 1 MB/s)
Maximale aggregierte Parallelität	Bis zu 50 gleichzeitige Aufrufe (5 Poller mal 10)

Beispiel — Zuordnung der Ereignisquelle im Warteschlangenmodus im On-Demand-Modus (AWS CLI)

Folgendes AWS Der CLI-Befehl erstellt eine Ereignisquellenzuordnung im Warteschlangenmodus im On-Demand-Modus. Ersetzen *my-function*, die Broker-Endpunkte, *my-queue-topic*, die Subnetz-IDs und die Sicherheitsgruppen-ID mit Ihren eigenen Werten.

```
PROMPT> aws lambda create-event-source-mapping \
  --function-name my-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":["broker1.us-
west-2.example.com:9092","broker2.us-west-2.example.com:9092"]}}' \
  --topics my-queue-topic \
  --batch-size 100 \
  --self-managed-kafka-event-source-config '{"ConsumptionMode": "Queue"}' \
  --source-access-configurations
 '[{"Type":"VPC_SUBNET","URI":"subnet-0123456789abcdef0"},
{"Type":"VPC_SUBNET","URI":"subnet-0123456789abcdef1"},
{"Type":"VPC_SECURITY_GROUP","URI":"sg-0123456789abcdef0"}]'
```

Modus bereitgestellter Kapazität

Verwenden Sie für Workloads mit strengen Leistungsanforderungen oder starken Datenverkehrsmustern den Bereitstellungsmodus. Im Bereitstellungsmodus haben Sie die direkte

Kontrolle über die Mindest- und Höchstanzahl von Event-Pollern, die für die Zuordnung der Ereignisquellen im Warteschlangenmodus vorgesehen sind.

Jeder Event-Poller im Warteschlangenmodus im bereitgestellten Modus unterstützt bis zu 5 MB/s des Durchsatzes und bis zu 10 gleichzeitige Lambda-Aufrufe.

In der folgenden Tabelle sind die Grenzwerte für den bereitgestellten Modus aufgeführt.

Limit	Wert
Durchsatz pro Event-Poller	Bis zu 5 MB/s
Gleichzeitige Aufrufe pro Event-Poller	Bis zu 10
Mindestanzahl von Event-Pollern (`MinimumPollers`)	1—200. Standard: 1
Maximale Anzahl von Event-Pollern (`MaximumPollers`)	1—2.000. Standard: 200
Zuordnungen der Ereignisquellen pro Poller-Gruppe	Bis zu 100
Aggregieren Sie die maximale Anzahl von Pollern pro Poller-Gruppe	2.000
Durchsatz pro EPU (aggregiert)	Bis zu 5 MB/s pro Poller; das Aggregat hängt von der Packungsdichte des Pollers ab

Der bereitgestellte Modus wird auf der Grundlage der Nutzung der Event Poller Unit (EPU) berechnet. Details zu den Preisen finden Sie unter AWS Lambda -Preise.

Beispiel — Zuordnung der Ereignisquelle im Warteschlangenmodus mit dem bereitgestellten Modus (AWS CLI)

Folgendes AWS Der CLI-Befehl erstellt eine Ereignisquellenzuordnung im Warteschlangenmodus im bereitgestellten Modus.

```
PROMPT> aws lambda create-event-source-mapping \
  --function-name my-function \
```

```
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":["broker1.us-
west-2.example.com:9092","broker2.us-west-2.example.com:9092"]}}' \
  --topics my-queue-topic \
  --batch-size 100 \
  --self-managed-kafka-event-source-config '{"ConsumptionMode": "Queue"}' \
  --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}' \
  --source-access-configurations
 '[{"Type":"VPC_SUBNET","URI":"subnet-0123456789abcdef0"},
{"Type":"VPC_SUBNET","URI":"subnet-0123456789abcdef1"},
{"Type":"VPC_SECURITY_GROUP","URI":"sg-0123456789abcdef0"}]'
```

Ersetzen Sie die folgenden Platzhalter durch Ihre eigenen Werte:

- *subnet-0123456789abcdef0* und *subnet-0123456789abcdef1*: Ihre Subnetz-IDs.
- *sg-0123456789abcdef0*: Ihre Sicherheitsgruppen-ID.
- `MinimumPollers` und `MaximumPollers`: entsprechend Ihren Durchsatzanforderungen einstellen.

> **Note**
>
> Um vom Bereitstellungsmodus zurück zum On-Demand-Modus zu wechseln, geben Sie ein leeres Feld ein `ProvisionedPollerConfig`:
>
> ```
> PROMPT> aws lambda update-event-source-mapping \
>   --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
>   --provisioned-poller-config '{}'
> ```

Auswahl des Skalierungsmodus

Überlegungen	On-demand	Bereitgestellt
Einrichtung	Keine zusätzliche Konfiguration	Konfiguriere `MinimumPollers` und `MaximumPollers`
Cost (Kosten)	Keine zusätzlichen EPU-Gebühren	Die EPU-Gebühren basieren auf der Nutzung pro Stunde

Überlegungen	On-demand	Bereitgestellt
Am besten geeignet für	Low-concurrency Arbeitsbelastungen	Strenge Latenzanforderungen, starker Datenverkehr
Gruppierung der Zuordnung der Ereignisquelle (`PollerGroupName`)	Nicht unterstützt	Unterstützt. Teilen Sie EPUs über Quelle-Quellzuordnungen von Queue-Ereignissen hinweg
Low-latency Verarbeitung	Nicht verfügbar	Available (Verfügbar)
Filterung der Schemaregistrierung (Avro/Protobuf)	Nicht verfügbar	Available (Verfügbar)

EPU teilt sich die Isolierung

Im bereitgestellten Modus können Sie mehrere Ereignisquellenzuordnungen innerhalb derselben VPC gruppieren, um die Kapazität der Event Poller Unit (EPU) gemeinsam zu nutzen, indem Sie ihnen dieselbe zuweisen `PollerGroupName`. Durch die Gruppierung der Ereignisquellenzuordnung können Ereignisquellenzuordnungen mit geringem Durchsatz auf gemeinsam genutzte EPUs gepackt werden, sodass nicht für jede einzelne Einheit dedizierte Kapazität erforderlich ist, wodurch die Kosten für den Bereitstellungsmodus gesenkt werden. Details hierzu finden Sie unter Kostenoptimierung für den Bereitstellungsmodus.

Zuordnungen von Ereignisquellen im Warteschlangenmodus verwenden EPUs nur gemeinsam mit anderen Zuordnungen von Ereignisquellen im Warteschlangenmodus. EIN `PollerGroupName` Die von einer Ereignisquellenzuordnung im Warteschlangenmodus verwendete Ereignisquellenzuordnung kann nicht mit einer Kafka Streams-Ereignisquellenzuordnung gemeinsam genutzt werden und umgekehrt. Wenn Sie eine konfigurieren `PollerGroupName` wird bereits vom anderen Verbrauchsmodus verwendet, `CreateEventSourceMapping` gibt einen Validierungsfehler zurück. Um die Gruppierung der Ereignisquellenzuordnung im Warteschlangenmodus zu verwenden, erstellen Sie eine `PollerGroupName` speziell für Ihre Quelle-Zuordnungen für Queue-Ereignisse reserviert.

Beispielereignis

Wenn Lambda Ihre Funktion mit einer Ereignisquelle im Warteschlangenmodus aufruft, ist die Nutzlast des Ereignisses identisch mit einem Standard-Kafka-Ereignis. Jeder Datensatz enthält das Thema, die Partition, den Offset, den Zeitstempel sowie den Base64-kodierten Schlüssel und Wert:

```
{
  "eventSource": "SelfManagedKafka",
  "bootstrapServers": "broker1.example.com:9092,broker2.example.com:9092",
  "records": {
    "my-queue-topic-0": [
      {
        "topic": "my-queue-topic",
        "partition": 0,
        "offset": 15,
        "timestamp": 1545084650987,
        "timestampType": "CREATE_TIME",
        "key": "abcDEFghiJKLmnoPQRstuVWXyz1234==",
        "value": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
        "headers": [
          {
            "headerKey": [104, 101, 97, 100, 101, 114, 86, 97, 108, 117, 101]
          }
        ]
      }
    ],
    "my-queue-topic-3": [
      {
        "topic": "my-queue-topic",
        "partition": 3,
        "offset": 42,
        "timestamp": 1545084651000,
        "timestampType": "CREATE_TIME",
        "key": "YW5vdGhlcktleQ==",
        "value": "VGhpcyBpcyBhbm90aGVyIG1lc3NhZ2U=",
        "headers": []
      }
    ]
  }
}
```

Idempotenz erforderlich

Lambda-Ereignisquellenzuordnungen verarbeiten jedes Ereignis mindestens einmal, sodass es zu einer doppelten Verarbeitung von Datensätzen kommen kann. Da der Warteschlangenmodus die Reihenfolge nicht garantiert, kann eine doppelte oder falsche Verarbeitung zu falschen Ergebnissen führen, z. B. zu Doppelzählungen oder einem inkonsistenten Zustand, wenn Ihr Funktionscode nicht idempotent ist. Machen Sie Ihren Funktionscode idempotent.

Einschränkungen

- Self-managed Nur Kafka-Cluster. Amazon MSK wird beim Start nicht unterstützt.
- Apache Kafka 4.2 oder höher erforderlich. Der Cluster muss das Share Consumer-Protokoll mit aktivierten Share-Gruppen unterstützen.
- Das Batching-Fenster wird nicht unterstützt. Das `MaximumBatchingWindowInSeconds` Der Parameter kann nicht mit Kafka Queue-Ereignisquellenzuordnungen verwendet werden.
- Die Startposition wird nicht unterstützt. `StartingPosition` und `StartingPositionTimestamp` kann nicht mit Ereignisquellenzuordnungen im Warteschlangenmodus verwendet werden. Die Share-Gruppe verbraucht von ihrer zugesagten Offset-Position aus.
- EPU-Isolierung. Zuordnungen von Ereignisquellen im Warteschlangenmodus können EPUs nicht gemeinsam mit Kafka Streams-Ereignisquellenzuordnungen verwenden. Verwenden Sie separate `PollerGroupName` Werte.

Anpassbare Verbrauchergruppen-ID in Lambda

Wenn Sie Amazon MSK oder selbstverwaltetes Apache Kafka als Quelle einrichten, können Sie eine Verbrauchergruppen-ID angeben. Diese Nutzergruppen-ID ist ein vorhandener Bezeichner für die Kafka-Verbrauchergruppe, der Ihre Lambda-Ereignisfunktion beitreten soll. Mit diesem Feature können Sie alle laufenden Kafka-Datensatzverarbeitungs-Setups nahtlos von anderen Verbrauchern auf Lambda migrieren.

Kafka verteilt Nachrichten an alle Verbraucher in einer Verbrauchergruppe. Wenn Sie eine Verbrauchergruppen-ID angeben, die andere aktive Verbraucher hat, empfängt Lambda nur einen Teil der Nachrichten aus dem Kafka-Thema. Wenn Sie möchten, dass Lambda alle Nachrichten im Thema verarbeitet, deaktivieren Sie alle anderen Verbraucher in dieser Verbrauchergruppe.

Wenn Sie außerdem eine Nutzergruppen-ID angeben und Kafka eine gültige bestehende Nutzergruppe mit derselben ID findet, ignoriert Lambda die StartingPositionfür Ihre Ereignisquellenzuordnung. Stattdessen beginnt Lambda mit der Verarbeitung von Datensätzen gemäß dem zugesagten Versatz der Konsumentengruppe. Wenn Sie eine Konsumentengruppen-ID angeben und Kafka keine vorhandene Konsumentengruppe finden kann, konfiguriert Lambda Ihre Ereignisquelle mit der angegebenen `StartingPosition`.

Die Konsumentengruppen-ID, die Sie angeben, muss unter all Ihren Kafka-Ereignisquellen eindeutig sein. Nachdem Sie eine Kafka-Ereignisquellenzuordnung mit der angegebenen Konsumentengruppen-ID erstellt haben, können Sie diesen Wert nicht aktualisieren.

Filtern von Ereignissen aus Amazon MSK und selbstverwalteten Apache-Kafka-Ereignisquellen

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Event-Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

> **Note**
>
> Zuordnungen von Ereignisquellen von Amazon MSK und selbstverwaltetem Apache Kafka unterstützen nur die Filterung nach dem Schlüssel `value`.

Themen

- Grundlagen der Kafka-Ereignisfilterung

Grundlagen der Kafka-Ereignisfilterung

Angenommen, ein Producer schreibt Nachrichten zu einem Thema in Ihrem Kafka-Cluster, entweder im gültigen JSON-Format oder als einfache Zeichenfolgen. Ein Beispieldatensatz würde wie folgt aussehen, wobei die Nachricht im `value`-Feld in eine Base64-kodierte Zeichenfolge umgewandelt wurde.

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Angenommen, Ihr Apache-Kafka-Producer schreibt Nachrichten zu Ihrem Thema im folgenden JSON-Format.

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

Sie können den `value`-Schlüssel verwenden, um Datensätze zu filtern. Angenommen, Sie möchten nur die Datensätze filtern, bei denen `device_ID` mit den Buchstaben AB beginnen. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer AWS SAM Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

Mit Kafka können Sie auch Datensätze filtern, bei denen die Nachricht eine einfache Zeichenfolge ist. Angenommen, Sie möchten die Meldungen ignorieren, in denen die Zeichenfolge „error“ ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
```

```
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer Vorlage hinzufügen. AWS SAM

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Kafka-Nachrichten müssen UTF-8 codierte Zeichenfolgen sein, entweder einfache Zeichenketten oder im JSON-Format. Das liegt daran, dass Lambda Kafka-Byte-Arrays dekodiert, UTF-8 bevor Filterkriterien angewendet werden. Wenn Ihre Nachrichten eine andere Kodierung verwenden, z. UTF-16 B. ASCII, oder wenn das Nachrichtenformat nicht dem `FilterCriteria` Format entspricht, verarbeitet Lambda nur Metadatenfilter. In der folgenden Tabelle ist das Verhalten zusammengefasst:

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
Einfache Zeichenfolge	Einfache Zeichenfolge	Lambda filtert basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Gültiges JSON	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Einfache Zeichenfolge	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenscha

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
		ften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Non-UTF-8 codierte Zeichenfolge	JSON, einfache Zeichenfolge oder kein Muster	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.

Verwenden von Schemaregistern mit Kafka-Ereignisquellen in Lambda

Schemaregister unterstützen Sie bei der Definition und Verwaltung von Datenstromschemas. Ein Schema definiert die Struktur und das Format eines Datensatzes. Im Kontext von Kafka-Ereignisquellenzuordnungen können Sie eine Schemaregistrierung konfigurieren, um die Struktur und das Format von Kafka-Nachrichten anhand vordefinierter Schemas zu validieren, bevor sie Ihre Lambda-Ereignisfunktion erreichen. Dadurch wird Ihrer Anwendung eine Ebene der Datenverwaltung hinzugefügt, und Sie können Datenformate effizient verwalten, die Schemakonformität sicherstellen und die Kosten durch Ereignisfilterung optimieren.

Dieses Feature ist mit allen Programmiersprachen kompatibel, jedoch sollten Sie folgende wichtige Punkte beachten:

- Powertools for Lambda bietet spezifische Unterstützung für Java, Python und TypeScript gewährleistet die Konsistenz mit bestehenden Kafka-Entwicklungsmustern und ermöglicht den direkten Zugriff auf Geschäftsobjekte ohne benutzerdefinierten Deserialisierungscode
- Dieses Feature ist nur für Zuordnungen von Ereignisquellen im Bereitstellungsmodus verfügbar. Die Schemaregistrierung unterstützt keine Zuordnungen von Ereignisquellen im On-Demand-Modus. Wenn Sie den Bereitstellungsmodus verwenden und eine Schemaregistrierung konfiguriert haben, können Sie nicht in den On-Demand-Modus wechseln, es sei denn, Sie entfernen zuvor Ihre Schemaregistrierungskonfiguration. Weitere Informationen finden Sie unter the section called “Modus bereitgestellter Kapazität”.
- Sie können nur eine Schemaregistrierung pro Zuordnung von Ereignisquellen (Event Source Mapping, (ESM) konfigurieren. Die Verwendung einer Schemaregistrierung mit Ihrer Kafka-Ereignisquelle kann die Nutzung Ihrer Lambda Event Poller Unit (EPU) erhöhen, was eine Preisdimension für den Bereitstellungsmodus darstellt.

Topics

- Optionen für die Schemaregistrierung
- So führt Lambda die Schemavalidierung für Kafka-Nachrichten durch
- Konfiguration einer Kafka-Schemaregistrierung
- Filterung nach Avro und Protobuf
- Nutzdatenformate und Deserialisierungsverhalten
- Arbeiten mit deserialisierten Daten in Lambda Event Functions
- Authentifizierungsmethoden für Ihre Schemaregistrierung

- Fehlerbehandlung und -behebung bei Problemen mit der Schemaregistrierung

Optionen für die Schemaregistrierung

Lambda unterstützt die folgenden Optionen für die Schemaregistrierung:

- AWS Glue Schemaregistrierung
- Confluent-Cloud-Schemaregistrierung
- Self-managed Konfluente Schemaregistrierung

Ihre Schemaregistrierung unterstützt die Validierung von Nachrichten in den folgenden Datenformaten:

- Apache Avro
- Protokollpuffer (Protobuf)
- JSON-Schema () JSON-SE

Um eine Schemaregistrierung zu verwenden, stellen Sie zunächst sicher, dass sich Ihre Zuordnung von Ereignisquellen im Bereitstellungsmodus befindet. Wenn Sie eine Schemaregistrierung verwenden, fügt Lambda Metadaten zum Schema zu den Nutzdaten hinzu. Weitere Informationen finden Sie unter Nutzdatenformate und Deserialisierungsverhalten.

So führt Lambda die Schemavalidierung für Kafka-Nachrichten durch

Wenn Sie eine Schemaregistrierung konfigurieren, führt Lambda für jede Kafka-Nachricht die folgenden Schritte aus:

1. Lambda fragt den Kafka-Datensatz aus Ihrem Cluster ab.
2. Lambda validiert ausgewählte Nachrichtenattribute im Datensatz anhand eines bestimmten Schemas in Ihrer Schemaregistrierung.
   - Wenn das der Nachricht zugeordnete Schema nicht in der Registrierung gefunden wird, sendet Lambda die Nachricht mit dem Ursachencode `SCHEMA_NOT_FOUND` an eine DLQ.
3. Lambda deserialisiert die Nachricht gemäß der Schemaregistrierungskonfiguration, um die Nachricht zu validieren. Wenn eine Ereignisfilterung konfiguriert ist, führt Lambda anschließend eine Filterung basierend auf den konfigurierten Filterkriterien durch.

- Wenn die Deserialisierung fehlschlägt, sendet Lambda die Nachricht mit dem Ursachencode `DESERIALIZATION_ERROR` an eine DLQ. Wenn keine DLQ konfiguriert ist, verwirft Lambda die Nachricht.

4. Wenn die Nachricht von der Schemaregistrierung validiert und nicht durch Ihre Filterkriterien herausgefiltert wird, ruft Lambda Ihre Funktion mit der Nachricht auf.

Dieses Feature dient zur Validierung von Nachrichten, die bereits mit Kafka-Clients erstellt wurden, die in eine Schemaregistrierung integriert sind. Wir empfehlen, Ihre Kafka-Producer so zu konfigurieren, dass sie mit Ihrer Schemaregistrierung zusammenarbeiten, um korrekt formatierte Nachrichten zu erstellen.

Konfiguration einer Kafka-Schemaregistrierung

Die folgenden Konsolenschritte fügen Ihrer Zuordnung von Ereignisquellen eine Kafka-Schemaregistrierungskonfiguration hinzu.

Hinzufügen einer Kafka-Schemaregistrierungskonfiguration zu Ihrer Zuordnung von Ereignisquellen (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie Konfiguration.
3. Wählen Sie Auslöser.
4. Wählen Sie die Kafka-Zuordnung von Ereignisquellen, für die Sie eine Schemaregistrierung konfigurieren möchten und wählen Sie Bearbeiten.
5. Wählen Sie unter Konfiguration des Event-Pollers die Option Schemaregistrierung konfigurieren aus. Ihre Zuordnung von Ereignisquellen muss sich im Bereitstellungsmodus befinden, damit diese Option angezeigt wird.
6. Geben Sie für Schema Registry URI den ARN Ihrer AWS Glue Schemaregistry oder die HTTPS-URL Ihrer Confluent Cloud-Schemaregistry oder Self-Managed Confluent Schema Registry ein.
7. Die folgenden Konfigurationsschritte teilen Lambda mit, wie auf Ihre Schemaregistrierung zugegriffen werden kann. Weitere Informationen finden Sie unter ???.

    - Wählen Sie für Zugriffskonfigurationstyp den Authentifizierungstyp aus, den Lambda für den Zugriff auf Ihre Schemaregistrierung verwendet.

- Geben Sie für URI der Zugriffskonfiguration die ARN des Secrets-Manager-Geheimnisses ein, um sich bei Ihrer Schemaregistrierung zu authentifizieren, falls zutreffend. Stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion die richtigen Berechtigungen enthält.

8. Das Feld Verschlüsselung gilt nur, wenn Ihre Schemaregistrierung von einer privaten Zertifizierungsstelle (CA) oder einer Zertifizierungsstelle (CA) signiert ist, die sich nicht im Lambda-Vertrauensspeicher befindet. Geben Sie gegebenenfalls den geheimen Schlüssel mit dem privaten CA-Zertifikat an, das von Ihrer Schemaregistrierung für die TLS-Verschlüsselung verwendet wird.
9. Wählen Sie unter Ereignisdatensatzformat aus, wie Lambda die Datensätze nach der Schemavalidierung an Ihre Funktion übermitteln soll. Weitere Informationen finden Sie unter Beispiele für das Nutzdatenformat.

    - Wenn Sie JSON wählen, liefert Lambda die Attribute, die Sie unten unter „Schemavalidierungsattribut“ auswählen, im Standard-JSON-Format. Die Attribute, die Sie nicht auswählen, liefert Lambda unverändert.
    - Wenn Sie SOURCE wählen, liefert Lambda die Attribute, die Sie unten unter „Schemavalidierungsattribut“ auswählen, im ursprünglichen Quellformat.

10. Wählen Sie für Schemavalidierungsattribut die Nachrichtenattribute aus, die Lambda mithilfe Ihrer Schemaregistrierung validieren und deserialisieren soll. Sie müssen mindestens entweder KEY oder VALUE auswählen. Wenn Sie JSON als Ereignisdatensatzformat ausgewählt haben, deserialisiert Lambda auch die ausgewählten Nachrichtenattribute, bevor sie an Ihre Funktion gesendet werden. Weitere Informationen finden Sie unter Nutzdatenformate und Deserialisierungsverhalten.
11. Wählen Sie Speichern.

Sie können auch die Lambda-API verwenden, um Ihre Zuordnung von Ereignisquellen mit einer Schemaregistrierungskonfiguration zu erstellen oder zu aktualisieren. Die folgenden Beispiele zeigen, wie Sie eine AWS Glue oder Confluent-Schemaregistry mithilfe der konfigurieren AWS CLI, was den API-Operationen UpdateEventSourceMappingund in der CreateEventSourceMappingAPI-Referenz entspricht:AWS Lambda

> **Important**
>
> Wenn Sie ein Konfigurationsfeld für die Schemaregistrierung mithilfe der AWS CLI oder der `update-event-source-mapping` API aktualisieren, müssen Sie alle Felder der Schemaregistrierungskonfiguration aktualisieren.

Create Event Source Mapping

```
aws lambda create-event-source-mapping \
  --function-name my-schema-validator-function \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \
  --topics my-kafka-topic \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --amazon-managed-kafka-event-source-mapping '{
      "SchemaRegistryConfig" : {
          "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
          "AccessConfigs": [{
              "Type": "BASIC_AUTH",
              "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
          }],
          "EventRecordFormat": "JSON",
          "SchemaValidationConfigs": [
          {
              "Attribute": "KEY"
          },
          {
              "Attribute": "VALUE"
          }]
      }
  }'
```

Update AWS Glue Schema Registry

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
registryName",
```

```
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry with Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "AccessConfigs": [{
                "Type": "BASIC_AUTH",
                "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
            }],
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry without Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
```

```
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Remove Schema Registry Configuration

Um eine Schemaregistrierungskonfiguration aus Ihrer Ereignisquellenzuordnung zu entfernen, können Sie den CLI-Befehl UpdateEventSourceMappingin der AWS Lambda API-Referenz verwenden.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {}
    }'
```

Filterung nach Avro und Protobuf

Wenn Sie die Formate Avro oder Protobuf mit einer Schemaregistrierung verwenden, können Sie die Ereignisfilterung auf Ihre Lambda-Ereignisfunktion anwenden. Die Filtermuster werden nach der Schemavalidierung auf die deserialisierte klassische JSON-Darstellung Ihrer Daten angewendet. Mit einem Avro-Schema, das Produktdetails einschließlich des Preises definiert, können Sie beispielsweise Nachrichten basierend auf dem Preiswert filtern:

> **Note**
>
> Bei der Deserialisierung wird Avro in Standard-JSON konvertiert, was bedeutet, dass es nicht direkt wieder in ein Avro-Objekt konvertiert werden kann. Wenn Sie eine Konvertierung in ein Avro-Objekt benötigen, verwenden Sie stattdessen das SOURCE-Format.
> Bei der Protobuf-Deserialisierung stimmen die Feldnamen im resultierenden JSON mit denen überein, die in Ihrem Schema definiert sind, anstatt wie bei Protobuf üblich in Camelcase konvertiert zu werden. Beachten Sie dies bei der Erstellung von Filtermustern.

```
aws lambda create-event-source-mapping \
    --function-name myAvroFunction \
    --topics myAvroTopic \
    --starting-position TRIM_HORIZON \
    --kafka-bootstrap-servers '["broker1:9092", "broker2:9092"]' \
    --schema-registry-config '{
        "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
myAvroRegistry",
        "EventRecordFormat": "JSON",
        "SchemaValidationConfigs": [
            {
                "Attribute": "VALUE"
            }
        ]
    }' \
    --filter-criteria '{
        "Filters": [
            {
                "Pattern": "{ \"value\" : { \"field_1\" : [\"value1\"], \"field_2\" :
 [\"value2\"] } }"
            }
        ]
    }'
```

In diesem Beispiel analysiert das Filtermuster das `value`-Objekt und gleicht Nachrichten in `field_1` mit `"value1"` und `field_2` mit `"value2"` ab. Die Filterkriterien werden anhand der deserialisierten Daten ausgewertet, nachdem Lambda die Nachricht vom Avro-Format in JSON konvertiert hat.

Ausführlichere Informationen zur Ereignisfilterung finden Sie unter Lambda-Ereignisfilterung.

Nutzdatenformate und Deserialisierungsverhalten

Bei Verwendung einer Schemaregistrierung liefert Lambda die endgültigen Nutzdaten in einem Format, das den regulären Ereignisnutzdaten ähnelt, jedoch einige zusätzliche Felder enthält. Die zusätzlichen Felder hängen vom Parameter `SchemaValidationConfigs` ab. Für jedes Attribut, das Sie für die Validierung auswählen (Schlüssel oder Wert), fügt Lambda den Nutzdaten entsprechende Schemametadaten hinzu.

> **Note**
>
> Sie müssen aws-lambda-java-events mindestens auf Version 3.16.0 aktualisieren, um Schemametadatenfelder verwenden zu können.

Wenn Sie beispielsweise das Feld `value` validieren, fügt Lambda Ihren Nutzdaten ein Feld namens `valueSchemaMetadata` hinzu. In ähnlicher Weise fügt Lambda für das Feld `key` ein Feld namens `keySchemaMetadata` hinzu. Diese Metadaten enthalten Informationen über das Datenformat und die Schema-ID, die für die Validierung verwendet werden:

```
"valueSchemaMetadata": {
    "dataFormat": "AVRO",
    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

Der Parameter `EventRecordFormat` kann entweder auf `JSON` oder `SOURCE` gesetzt werden. Damit wird festgelegt, wie Lambda schemavalidierte Daten verarbeitet, bevor sie an Ihre Funktion übermittelt werden. Jede Option bietet unterschiedliche Verarbeitungsmöglichkeiten:

- `JSON` – Lambda deserialisiert die validierten Attribute in das Standard-JSON-Format, sodass die Daten direkt in Sprachen mit nativer JSON-Unterstützung verwendet werden können. Dieses Format ist ideal, wenn Sie das ursprüngliche Binärformat nicht beibehalten oder mit generierten Klassen arbeiten müssen.
- `SOURCE`- Lambda behält das ursprüngliche Binärformat der Daten als Base64-encoded Zeichenfolge bei und ermöglicht so eine direkte Konvertierung in Avro- oder Protobuf-Objekte. Dieses Format ist unerlässlich, wenn Sie mit stark typisierten Sprachen arbeiten oder die volle Leistungsfähigkeit von Avro- oder Protobuf-Schemas beibehalten müssen.

Basierend auf diesen Formatmerkmalen und sprachspezifischen Überlegungen empfehlen wir die folgenden Formate:

Empfohlene Formate basierend auf der Programmiersprache

Sprache	Avro	Protobuf	JSON
Java	SOURCE	SOURCE	SOURCE

Sprache	Avro	Protobuf	JSON
Python	JSON	JSON	JSON
NodeJS	JSON	JSON	JSON
.NET	SOURCE	SOURCE	SOURCE
Weitere	JSON	JSON	JSON

In den folgenden Abschnitten werden diese Formate ausführlich beschrieben und Beispielnutzdaten für jedes Format bereitgestellt.

JSON-Format

Wenn Sie dies wählen `JSONEventRecordFormat`, validiert und deserialisiert Lambda die Nachrichtenattribute, die Sie in dem `SchemaValidationConfigs` Feld ausgewählt haben (die Attribute). `key` and/or `value` Lambda liefert diese ausgewählten Attribute als Base64-codierte Zeichenfolgen ihrer Standard-JSON-Darstellung in Ihrer Funktion.

> **Note**
>
> Bei der Deserialisierung wird Avro in Standard-JSON konvertiert, was bedeutet, dass es nicht direkt wieder in ein Avro-Objekt konvertiert werden kann. Wenn Sie eine Konvertierung in ein Avro-Objekt benötigen, verwenden Sie stattdessen das SOURCE-Format.
> Bei der Protobuf-Deserialisierung stimmen die Feldnamen im resultierenden JSON mit denen überein, die in Ihrem Schema definiert sind, anstatt wie bei Protobuf üblich in Camelcase konvertiert zu werden. Beachten Sie dies bei der Erstellung von Filtermustern.

Im Folgenden finden Sie ein Beispiel für Nutzdaten, wobei davon ausgegangen wird, dass Sie `JSON` als `EventRecordFormat` und sowohl die `key`- als auch die `value`-Attribute als `SchemaValidationConfigs` wählen.

```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
```

```
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==", //Base64 encoded string of JSON
            "keySchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "value":"abcDEFghiJKLmnoPQRstuVWXyz1234", //Base64 encoded string of JSON
            "valueSchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

In diesem Beispiel:

- Sowohl `key` als auch `value` sind Base64-codierte Zeichenfolgen ihrer JSON-Darstellung nach der Deserialisierung.
- Lambda enthält Schemametadaten für beide Attribute in `keySchemaMetadata` und`valueSchemaMetadata`.
- Ihre Funktion kann die `key`- und `value`-Zeichenfolgen decodieren, um auf die deserialisierten JSON-Daten zuzugreifen.

Das JSON-Format wird für Sprachen empfohlen, die nicht stark typisiert sind, wie Python oder Node.js. Diese Sprachen bieten native Unterstützung für die Konvertierung von JSON in Objekte.

Quellformat

Wenn Sie `SOURCE` als `EventRecordFormat` auswählen, validiert Lambda den Datensatz trotzdem anhand der Schemaregistrierung, liefert jedoch die ursprünglichen Binärdaten ohne Deserialisierung an Ihre Funktion. Diese Binärdaten werden als Base64-codierte Zeichenfolge der ursprünglichen Byte-Daten geliefert, wobei die vom Producer angehängten Metadaten entfernt werden. Dadurch können Sie die binären Rohdaten direkt in Avro- und Protobuf-Objekte innerhalb Ihres Funktionscodes konvertieren. Wir empfehlen die Verwendung von Powertools for AWS Lambda, das die rohen Binärdaten deserialisiert und Ihnen Avro- und Protobuf-Objekte direkt zur Verfügung stellt.

Wenn Sie beispielsweise Lambda so konfigurieren, dass sowohl die `key`- als auch die `value`-Attribute validiert werden, aber das `SOURCE`-Format verwenden, erhält Ihre Funktion Nutzdaten wie diese:

```
{
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
    "bootstrapServers": "b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
    "records": {
        "mytopic-0": [
            {
                "topic": "mytopic",
                "partition": 0,
```

```
                "offset": 15,
                "timestamp": 1545084650987,
                "timestampType": "CREATE_TIME",
                "key": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string of
 Original byte data, producer-appended metadata removed
                "keySchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "value": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string
 of Original byte data, producer-appended metadata removed
                "valueSchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "headers": [
                    {
                        "headerKey": [
                            104,
                            101,
                            97,
                            100,
                            101,
                            114,
                            86,
                            97,
                            108,
                            117,
                            101
                        ]
                    }
                ]
            }
        ]
    }
}
```

In diesem Beispiel:

- Sowohl `key` als auch `value` enthalten die ursprünglichen Binärdaten als Base64-codierte Zeichenfolgen.
- Ihre Funktion muss die Deserialisierung mit den entsprechenden Bibliotheken verarbeiten.

`EventRecordFormat`Es wird empfohlen, `SOURCE` für zu wählen, wenn Sie Oder-Objekte verwenden, insbesondere bei Avro-generated Java-Funktionen Protobuf-generated . Dies liegt daran, dass Java stark typisiert ist und spezifische Deserialisierer für die Formate Avro und Protobuf erfordert. In Ihrem Funktionscode können Sie Ihre bevorzugte Avro- oder Protobuf-Bibliothek verwenden, um die Daten zu deserialisieren.

Arbeiten mit deserialisierten Daten in Lambda Event Functions

Powertools for AWS Lambda hilft Ihnen dabei, Kafka-Datensätze in Ihrem Funktionscode auf der Grundlage des von Ihnen verwendeten Formats zu deserialisieren. Dieses Dienstprogramm vereinfacht die Arbeit mit Kafka-Datensätzen, indem es die Datenkonvertierung übernimmt und gebrauchsfertige Objekte bereitstellt.

Um Powertools for AWS Lambda in Ihrer Funktion zu verwenden, müssen Sie Powertools for AWS Lambda entweder als Ebene oder als Abhängigkeit hinzufügen, wenn Sie Ihre Lambda-Ereignisfunktion erstellen. Anweisungen zur Einrichtung und weitere Informationen finden Sie in der AWS Lambda Powertools-Dokumentation für Ihre bevorzugte Sprache:

- Powertools für AWS Lambda (Java)
- Powertools für AWS Lambda (Python)
- Elektrowerkzeuge für AWS Lambda () TypeScript
- Elektrowerkzeuge für AWS Lambda (.NET)

> **Note**
>
> Bei der Arbeit mit der Schemaregistrierungsintegration können Sie zwischen dem `SOURCE`- und dem `JSON`-Format wählen. Jede Option unterstützt verschiedene Serialisierungsformate, wie unten gezeigt:

Format	Unterstützt
SOURCE	Avro und Protobuf (unter Verwendung der Lambda-Schemaregistrierungsintegration)
JSON	JSON-Daten

Wenn Sie das JSON Format SOURCE oder verwenden, können Sie Powertools for verwenden, AWS um die Daten in Ihrem Funktionscode zu deserialisieren. Hier sind Beispiele für den Umgang mit verschiedenen Datenformaten:

AVRO

Java-Beispiel:

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.avro.AvroProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class AvroDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, AvroProduct>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(AvroDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_AVRO)
    public String handleRequest(ConsumerRecords<String, AvroProduct> records,
 Context context) {
        for (ConsumerRecord<String, AvroProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            AvroProduct product = consumerRecord.value();
            LOGGER.info("AvroProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }
```

```
        return "OK";
    }

}
```

Python-Beispiel:

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer
from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

value_schema_str = open("customer_profile.avsc", "r").read()

schema_config = SchemaConfig(
value_schema_type="AVRO",
value_schema=value_schema_str)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript Beispiel:

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';

import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();
```

```
type Value = {
    id: number;
    name: string;
    price: number;
};

const schema = '{
    "type": "record",
    "name": "Product",
    "fields": [
        { "name": "id", "type": "int" },
        { "name": "name", "type": "string" },
        { "name": "price", "type": "double" }
    ]
}';

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'avro',
            schema: schema,
        },
    }
);
```

.NET-Beispiel:

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Avro;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda Event Function's Kafka event to be
 converted into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaAvroSerializer))]
```

```
namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

PROTOBUF

Java-Beispiel:

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.protobuf.ProtobufProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
```

```
import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class ProtobufDeserializationFunction
        implements RequestHandler<ConsumerRecords<String, ProtobufProduct>, String>
 {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(ProtobufDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_PROTOBUF)
    public String handleRequest(ConsumerRecords<String, ProtobufProduct> records,
 Context context) {
        for (ConsumerRecord<String, ProtobufProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            ProtobufProduct product = consumerRecord.value();
            LOGGER.info("ProtobufProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }

}
```

Python-Beispiel:

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger
```

```
from user_pb2 import User # protobuf generated class

logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(
value_schema_type="PROTOBUF",
value_schema=User)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript Beispiel:

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';
import { Product } from './product.generated.js';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
        }
    },
    {
        value: {
            type: 'protobuf',
            schema: Product,
```

```
        },
    }
);
```

.NET-Beispiel:

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Protobuf;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaProtobufSerializer))]

namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

JSON

Java-Beispiel:

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class JsonDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, Product>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(JsonDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_JSON)
    public String handleRequest(ConsumerRecords<String, Product> consumerRecords,
 Context context) {
        for (ConsumerRecord<String, Product> consumerRecord : consumerRecords) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            Product product = consumerRecord.value();
            LOGGER.info("Product: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }
}
```

Python-Beispiel:

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(value_schema_type="JSON")

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript Beispiel:

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
```

```
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'json',
        },
    }
);
```

.NET-Beispiel:

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Json;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaJsonSerializer))]

namespace JsonClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }
```

```
            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

Authentifizierungsmethoden für Ihre Schemaregistrierung

Um eine Schemaregistrierung verwenden zu können, muss Lambda in der Lage sein, sicher darauf zuzugreifen. Wenn Sie mit einer AWS Glue Schemaregistrierung arbeiten, stützt sich Lambda auf die IAM-Authentifizierung. Das bedeutet, dass die Ausführungsrolle Ihrer Funktion über die folgenden Berechtigungen für den Zugriff auf die AWS Glue Registrierung verfügen muss:

- GetRegistryin der AWS Glue Web-API-Referenz
- GetSchemaVersionin der AWS Glue Web-API-Referenz

Beispiel für die erforderliche IAM-Richtlinie:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:GetRegistry",
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

> **Note**
>
> Wenn Sie für AWS Glue Schemaregister eine AWS Glue Registrierung angeben`AccessConfigs`, gibt Lambda eine Validierungsausnahme zurück.

Wenn Sie mit einer Confluent-Schemaregistry arbeiten, können Sie eine von vier unterstützten Authentifizierungsmethoden für den `Type` Parameter Ihres Objekts wählen: KafkaSchemaRegistryAccessConfig

- BASIC_AUTH – Lambda verwendet die Authentifizierung mit Benutzername und Passwort oder API-Schlüssel und API-Geheimnis, um auf Ihre Registrierung zuzugreifen. Wenn Sie diese Option auswählen, geben Sie die Secrets-Manager-ARN mit Ihren Anmeldeinformationen im Feld „URI“ an.
- CLIENT_CERTIFICATE_TLS_AUTH – Lambda verwendet die gegenseitige TLS-Authentifizierung mit Client-Zertifikaten. Um diese Option zu verwenden, benötigt Lambda Zugriff sowohl auf das Zertifikat als auch auf den privaten Schlüssel. Geben Sie den Secrets-Manager-ARN mit diesen Anmeldeinformationen im URI-Feld an.
- OAUTHBEARER_AUTH — Lambda fordert ein Zugriffstoken von Ihrem OAuth 2.0-Identitätsanbieter an und sendet es als Bearer-Token an Ihre Registry. Lambda aktualisiert das Token, bevor es abläuft. Wenn Sie diese Option wählen, geben Sie den Secrets Manager Manager-ARN, der Ihr OAuth-Geheimnis enthält, in das URI-Feld ein. Das Secret verwendet dieselben Felder wie das Secret für die Broker-Authentifizierung und unterstützt dieselben beiden Grant-Typen. Den Inhalt des Geheimnisses finden Sie unterthe section called “Konfiguration des OAuth-Geheimnisses”.
- NO_AUTH – Das öffentliche CA-Zertifikat muss von einer Zertifizierungsstelle (CA) signiert sein, die sich im Lambda-Vertrauensspeicher befindet. Für ein privates CA/self-signed Zertifikat konfigurieren Sie das Root-CA-Zertifikat des Servers. Um diese Option zu verwenden, lassen Sie den Parameter `AccessConfigs` weg.

Wenn Lambda Zugriff auf ein privates CA-Zertifikat benötigt, um das TLS-Zertifikat Ihrer Schemaregistrierung zu überprüfen, wählen Sie zusätzlich `SERVER_ROOT_CA_CERT` als `Type` und geben Sie den Secrets-Manager-ARN für das Zertifikat in das URI-Feld ein.

> **Note**
>
> Um die Option `SERVER_ROOT_CA_CERT` in der Konsole zu konfigurieren, geben Sie den Geheimnis-ARN, der das Zertifikat enthält, in das Feld Verschlüsselung ein.

Die Authentifizierungskonfiguration für Ihre Schemaregistrierung ist unabhängig von der Authentifizierung, die Sie für Ihren Kafka-Cluster konfiguriert haben. Sie müssen beide separat konfigurieren, auch wenn sie ähnliche Authentifizierungsmethoden verwenden.

Fehlerbehandlung und -behebung bei Problemen mit der Schemaregistrierung

Wenn Sie eine Schemaregistrierung mit Ihrer Amazon MSK-Ereignisquelle verwenden, können verschiedene Fehler auftreten. Dieser Abschnitt enthält Anleitungen zu häufigen Problemen und deren Behebung.

Konfigurationsfehler

Diese Fehler treten beim Einrichten Ihrer Schemaregistrierungskonfiguration auf.

Bereitstellungsmodus erforderlich

- Fehlermeldung: `SchemaRegistryConfig is only available for Provisioned Mode. To configure Schema Registry, please enable Provisioned Mode by specifying MinimumPollers in ProvisionedPollerConfig.`
- Lösung: Aktivieren Sie den Bereitstellungsmodus für Ihre Zuordnung von Ereignisquellen, indem Sie den Parameter `MinimumPollers` in `ProvisionedPollerConfig` konfigurieren.

Ungültige Schemaregistrierungs-URL

- Fehlermeldung: `Malformed SchemaRegistryURI provided. Please provide a valid URI or ARN. For example, https://schema-registry.example.com:8081 or arn:aws:glue:us-east-1:123456789012:registry/ExampleRegistry.`
- Lösung: Geben Sie eine gültige HTTPS-URL für Confluent Schema Registry oder einen gültigen ARN für AWS Glue Schema Registry an.

Ungültiges oder fehlendes Ereignisdatensatzformat

- Fehlermeldung: `EventRecordFormat is a required field for SchemaRegistryConfig. Please provide one of supported format types: SOURCE, JSON.`

Lösung: Geben Sie EventRecordFormat in Ihrer Schemaregistrierungskonfiguration entweder SOURCE oder JSON an.

Doppelte Validierungsattribute

Fehlermeldung: `Duplicate KEY/VALUE Attribute in SchemaValidationConfigs. SchemaValidationConfigs must contain at most one KEY/VALUE Attribute.`

Lösung: Entfernen Sie doppelte KEY- oder VALUE-Attribute aus Ihrem SchemaValidationConfigs. Jeder Attributtyp darf nur einmal vorkommen.

Fehlende Validierungskonfiguration

Fehlermeldung: `SchemaValidationConfigs is a required field for SchemaRegistryConfig.`

Lösung: Fügen Sie SchemaValidationConfigs es zu Ihrer Konfiguration hinzu und geben Sie mindestens ein Validierungsattribut (KEY oder VALUE) an.

Zugriffs- und Berechtigungsfehler

Diese Fehler treten auf, wenn Lambda aufgrund von Berechtigungs- oder Authentifizierungsproblemen nicht auf die Schemaregistrierung zugreifen kann.

AWS Glue Zugriff auf die Schemaregistrierung verweigert

Fehlermeldung: `Cannot access Glue Schema with provided role. Please ensure the provided role can perform the GetRegistry and GetSchemaVersion Actions on your schema.`

Lösung: Fügen Sie die erforderlichen Berechtigungen (`glue:GetRegistry` und `glue:GetSchemaVersion`) zur Ausführungsrolle Ihrer Funktion hinzu.

Zugriff auf Confluent-Schemaregistrierung verweigert

Fehlermeldung: `Cannot access Confluent Schema with the provided access configuration.`

Lösung: Stellen Sie sicher, dass Ihre Authentifizierungsanmeldeinformationen (gespeichert in Secrets Manager) korrekt sind und über die erforderlichen Berechtigungen für den Zugriff auf die Schemaregistrierung verfügen.

Cross-account AWS Glue Schemaregistrierung

Fehlermeldung: `Cross-account Glue Schema Registry ARN not supported.`

Lösung: Verwenden Sie eine AWS Glue Schemaregistrierung, die sich in demselben AWS Konto wie Ihre Lambda-Ereignisfunktion befindet.

Cross-region AWS Glue Schemaregistrierung

Fehlermeldung: `Cross-region Glue Schema Registry ARN not supported.`

Lösung: Verwenden Sie eine AWS Glue Schemaregistrierung, die sich in derselben Region wie Ihre Lambda-Ereignisfunktion befindet.

Probleme beim Zugriff auf Geheimnisse

Fehlermeldung: `Lambda received InvalidRequestException from Secrets Manager.`

Lösung: Stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion berechtigt ist, auf das Geheimnis zuzugreifen, und dass das Geheimnis nicht mit einem AWS KMS Standardschlüssel verschlüsselt ist, wenn Sie von einem anderen Konto aus zugreifen.

Verbindungsfehler

Diese Fehler treten auf, wenn Lambda keine Verbindung mit der Schemaregistrierung herstellen kann.

Probleme mit der VPC-Konnektivität

Fehlermeldung: `Cannot connect to your Schema Registry. Your Kafka cluster's VPC must be able to connect to the schema registry. You can provide access by configuring AWS PrivateLink or a NAT Gateway or VPC Peering between Kafka Cluster VPC and the schema registry VPC.`

Lösung: Konfigurieren Sie Ihr VPC-Netzwerk so, dass Verbindungen zur Schemaregistry über AWS PrivateLink ein NAT-Gateway oder VPC-Peering möglich sind.

TLS-Handshake-Fehler

Fehlermeldung: `Unable to establish TLS handshake with the schema registry. Please provide correct CA-certificate or client certificate using Secrets Manager to access your schema registry.`

Lösung: Überprüfen Sie, ob Ihre CA-Zertifikate und Client-Zertifikate (für mTLS) korrekt sind und ordnungsgemäß in Secrets Manager konfiguriert wurden.

Drosselung

Fehlermeldung: `Receiving throttling errors when accessing the schema registry. Please increase API TPS limits for your schema registry.`

Lösung: Erhöhen Sie die API-Ratenlimits für Ihre Schemaregistrierung oder reduzieren Sie die Anzahl der Anforderungen von Ihrer Anwendung.

Self-managed Fehler in der Schema-Registrierung

Fehlermeldung: `Lambda received an internal server an unexpected error from the provided self-managed schema registry.`

Lösung: Überprüfen Sie den Zustand und die Konfiguration des Servers Ihrer selbstverwalteten Schemaregistrierung.

Verarbeitung mit niedriger Latenz für Kafka-Ereignisquellen

AWS Lambda unterstützt nativ die Ereignisverarbeitung mit niedriger Latenz für Anwendungen, die konsistente End-to-End-Latenzen von weniger als 100 Millisekunden erfordern. Auf dieser Seite finden Sie Konfigurationsdetails und Empfehlungen zur Aktivierung von Workflows mit geringer Latenz.

Verarbeitung mit geringer Latenz aktivieren

Um die Verarbeitung mit geringer Latenz für eine Kafka-Zuordnung von Ereignisquellen zu aktivieren, ist die folgende Grundkonfiguration erforderlich:

- Aktivieren Sie den Bereitstellungsmodus. Weitere Informationen finden Sie unter Modus bereitgestellter Kapazität.
- Setzen Sie den Parameter `MaximumBatchingWindowInSeconds` der Zuordnung von Ereignisquellen auf 0. Weitere Informationen finden Sie unter Batching-Verhalten.

Fine-tuning Ihr Kafka ESM mit niedriger Latenz

Beachten Sie die folgenden Empfehlungen, um Ihre Kafka-Zuordnung von Ereignisquellen für geringe Latenz zu optimieren:

Konfiguration des Bereitstellungsmodus

Im Bereitstellungsmodus für die Zuordnung von Kafka-Ereignisquellen können Sie den Durchsatz Ihrer Ereignisquellenzuordnung optimieren, indem Sie eine Mindest- und Höchstzahl von Ressourcen konfigurieren, die als Event Poller bezeichnet werden. Ein Event Poller (oder ein Poller) stellt eine Rechenressource dar, die einer Zuordnung von Ereignisquellen im bereitgestellten Modus zugrunde liegt und bis zu 5 Durchsätze zuweist. MB/s Jeder Ereignis-Poller unterstützt bis zu 5 gleichzeitige Lambda-Aufrufe.

Um die optimale Poller-Konfiguration für Ihre Anwendung zu ermitteln, berücksichtigen Sie Ihre maximale Erfassungsrate und Ihre Verarbeitungsanforderungen. Sehen wir uns ein vereinfachtes Beispiel an:

Bei einer Batchgröße von 20 Datensätzen und einer durchschnittlichen Zieldauer von 50 ms kann jeder Poller 2.000 Datensätze pro Sekunde verarbeiten, wobei eine Grenze von 5 eingehalten wird. MB/s Dies wird wie folgt berechnet: (20 Datensätze × 1000ms/50ms) × 5 gleichzeitige Lambda-Aufrufe. Wenn Ihre gewünschte Spitzenerfassungsrate also 20 000 Datensätze pro Sekunde beträgt, benötigen Sie mindestens 10 Ereignis-Poller.

> **Note**
>
> Wir empfehlen, zusätzliche Ereignis-Pollers als Puffer bereitzustellen, um einen konstanten Betrieb bei maximaler Kapazität zu vermeiden.

Der Bereitstellungsmodus skaliert Ihre Ereignis-Poller automatisch basierend auf den Datenverkehrsmustern innerhalb der konfigurierten minimalen und maximalen Ereignis-Poller, was eine Neugewichtung auslösen und somit zusätzliche Latenz verursachen kann. Sie können Auto Scaling deaktivieren, indem Sie denselben Wert für den minimalen und maximalen Ereignis-Poller konfigurieren.

Weitere Überlegungen

Zu den weiteren Überlegungen gehören:

- Kaltstarts durch den Aufruf Ihrer Lambda-Zielfunktion können möglicherweise die End-to-End-Latenz erhöhen. Um dieses Risiko zu verringern, sollten Sie in Erwägung ziehen, die bereitgestellte Parallelität oder die Zielfunktion Ihrer SnapStartEreignisquellenzuordnung zu aktivieren. Optimieren Sie außerdem die Speicherzuweisung Ihrer Funktion, um eine konsistente und optimale Ausführung sicherzustellen.
- Wenn auf 0 gesetzt `MaximumBatchingWindowInSeconds` ist, verarbeitet Lambda sofort alle verfügbaren Datensätze, ohne darauf zu warten, dass die gesamte Batchgröße gefüllt wird. Wenn Ihre Batchgröße beispielsweise auf 1.000 Datensätze festgelegt ist, aber nur 100 Datensätze verfügbar sind, verarbeitet Lambda diese 100 Datensätze sofort, anstatt darauf zu warten, dass sich die gesamten 1.000 Datensätze ansammeln.

> **Important**
>
> Die optimale Konfiguration für eine Verarbeitung mit geringer Latenz hängt stark von Ihrer spezifischen Workload ab. Wir empfehlen dringend, verschiedene Konfigurationen mit Ihrer tatsächlichen Workload zu testen, um die besten Einstellungen für Ihren Anwendungsfall zu ermitteln.

Konfiguration der Fehlerbehandlungssteuerungen für Kafka-Ereignisquellen

Sie können konfigurieren, wie Lambda Fehler und Wiederholungen für Ihre Kafka-Ereignisquellenzuordnungen behandelt. Mit diesen Konfigurationen können Sie steuern, wie Lambda fehlgeschlagene Datensätze verarbeitet und das Wiederholungsverhalten verwaltet.

Verfügbare Wiederholungskonfigurationen

Die folgenden Wiederholungskonfigurationen sind sowohl für Amazon MSK als auch für selbstverwaltete Kafka-Ereignisquellen verfügbar:

- Maximale Wiederholungsversuche — Die maximale Anzahl von Wiederholungsversuchen von Lambda, wenn Ihre Event-Funktion einen Fehler zurückgibt. Der erste Aufrufversuch wird dabei nicht mitgezählt. Die Standardeinstellung ist -1 (unendlich). Wenn Sie sowohl unendliche Wiederholungsversuche als auch ein Ziel für den Fall eines Fehlers konfigurieren, wendet Lambda automatisch maximal 10 Wiederholungsversuche an.
- Maximales Datensatzalter — Das maximale Alter eines Datensatzes, den Lambda an Ihre Funktion sendet. Die Standardeinstellung ist -1 (unendlich).
- Batch bei Fehler aufteilen — Wenn Ihre Funktion einen Fehler zurückgibt, teilen Sie den Batch in zwei kleinere Batches auf und versuchen Sie es jeweils einzeln erneut. Dies hilft, problematische Datensätze zu isolieren.
- Partielle Batch-Antwort — Erlauben Sie Ihrer Funktion, Informationen darüber zurückzugeben, bei welchen Datensätzen in einem Batch die Verarbeitung fehlgeschlagen ist, sodass Lambda nur die fehlgeschlagenen Datensätze erneut versuchen kann.

Konfiguration von Steuerelementen zur Fehlerbehandlung (Konsole)

Sie können das Wiederholungsverhalten konfigurieren, wenn Sie eine Kafka-Ereignisquellenzuordnung in der Lambda-Konsole erstellen oder aktualisieren.

So konfigurieren Sie das Wiederholungsverhalten für eine Kafka-Ereignisquelle (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Führen Sie eine der folgenden Aktionen aus:
    - Um einen neuen Kafka-Trigger hinzuzufügen, wählen Sie unter Funktionsübersicht die Option Trigger hinzufügen aus.

- Um einen vorhandenen Kafka-Trigger zu ändern, wählen Sie den Auslöser aus und klicken Sie dann auf Bearbeiten.

4. Wählen Sie unter Konfiguration des Event-Pollers den Bereitstellungsmodus aus, um die Kontrollen zur Fehlerbehandlung zu konfigurieren:

   a. Geben Sie für Wiederholungsversuche die maximale Anzahl von Wiederholungsversuchen ein (0-10000 oder -1 für unendlich).

   b. Geben Sie für Maximales Datensatzalter das Höchstalter in Sekunden ein (60-604800 oder -1 für unendlich).

   c. Um die Stapelaufteilung bei Fehlern zu aktivieren, wählen Sie Batch bei Fehler aufteilen aus.

   d. Um die teilweise Batch-Antwort zu aktivieren, wählen Sie ReportBatchItemFailures.

5. Wählen Sie „Hinzufügen“ oder „Speichern“.

Konfiguration des Wiederholungsverhaltens (AWS CLI)

Verwenden Sie die folgenden AWS CLI Befehle, um das Wiederholungsverhalten für Ihre Kafka-Ereignisquellenzuordnungen zu konfigurieren.

Erstellen einer Ereignisquellenzuordnung mit Wiederholungskonfigurationen

Im folgenden Beispiel wird eine selbstverwaltete Kafka-Ereignisquellenzuordnung mit Steuerelementen zur Fehlerbehandlung erstellt:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics my-kafka-topic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

Für Amazon MSK-Ereignisquellen:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSMSKKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]' \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

Wiederholungskonfigurationen werden aktualisiert

Verwenden Sie den `update-event-source-mapping` Befehl, um Wiederholungskonfigurationen für eine bestehende Ereignisquellenzuordnung zu ändern:

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --maximum-retry-attempts 5 \
  --maximum-record-age-in-seconds 7200 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

PartialBatchResponse

Die partielle Batch-Antwort, auch bekannt als ReportBatchItemFailures, ist eine wichtige Funktion für die Fehlerbehandlung bei der Integration von Lambda mit Kafka-Quellen. Ohne diese Funktion führt ein Fehler, der in einem der Elemente eines Batches auftritt, dazu, dass alle Nachrichten in diesem Batch erneut verarbeitet werden. Wenn die partielle Batch-Antwort aktiviert und implementiert ist, gibt der Handler nur Identifikatoren für die fehlgeschlagenen Nachrichten zurück, sodass Lambda nur diese spezifischen Elemente erneut versuchen kann. Dies ermöglicht eine bessere Kontrolle darüber, wie Batches mit fehlgeschlagenen Nachrichten verarbeitet werden.

Verwenden Sie dieses JSON-Schema, um Batchfehler zu melden:

```
{
  "batchItemFailures": [
    {
```

```
      "itemIdentifier": {
        "partition": "topic-partition_number",
        "offset": 100
      }
    },
    ...
  ]
}
```

> **Important**
>
> Wenn Sie einen leeren, gültigen JSON-Code oder einen Nullwert zurückgeben, betrachtet die Zuordnung der Ereignisquelle einen Batch als erfolgreich verarbeitet. Jede zurückgegebene ungültige Topic-Partition_Number oder Offset, die im aufgerufenen Ereignis nicht enthalten waren, wird als Fehler behandelt, und der gesamte Batch wird erneut versucht.

Die folgenden Codebeispiele zeigen, wie eine partielle Batch-Antwort für Lambda-Ereignisfunktionen implementiert wird, die Ereignisse aus Kafka-Quellen empfangen. Die Funktion meldet die Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

Hier ist eine Python-Lambda-Handler-Implementierung, die diesen Ansatz zeigt:

```
import base64
from typing import Any, Dict, List

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, List[Dict[str,
 Dict[str, Any]]]]:
    failures: List[Dict[str, Dict[str, Any]]] = []
    records_dict = event.get("records", {})

    for topic_partition, records_list in records_dict.items():
        for record in records_list:
            topic = record.get("topic")
            partition = record.get("partition")
            offset = record.get("offset")
            value_b64 = record.get("value")

            try:
                data = base64.b64decode(value_b64).decode("utf-8")
                process_message(data)
            except Exception as exc:
```

```
                print(f"Failed to process record topic={topic} partition={partition}
 offset={offset}: {exc}")
                item_identifier: Dict[str, Any] = {
                    "partition": f"{topic}-{partition}",
                    "offset": int(offset) if offset is not None else None,
                }
                failures.append({"itemIdentifier": item_identifier})

    return {"batchItemFailures": failures}

def process_message(data: str) -> None:
    # Your business logic for a single message
    pass
```

Hier ist eine Node.js Version:

```
const { Buffer } = require("buffer");

const handler = async (event) => {
  const failures = [];

  for (let topicPartition in event.records) {
    const records = event.records[topicPartition];

    for (const record of records) {
      const topic = record.topic;
      const partition = record.partition;
      const offset = record.offset;
      const valueBase64 = record.value;
      const data = Buffer.from(valueBase64, "base64").toString("utf8");

      try {
        await processMessage(data);
      } catch (error) {
        console.error("Failed to process record", { topic, partition, offset, error });
        const itemIdentifier = {
          "partition": `${topic}-${partition}`,
          "offset": Number(offset),
        };
        failures.push({ itemIdentifier });
      }
    }
  }
```

```
  return { batchItemFailures: failures };
};

async function processMessage(payload) {
  // Your business logic for a single message
}

module.exports = { handler };
```

Hier ist eine Java-Version:

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.ArrayList;
import java.util.Base64;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class KafkaBatchHandler implements RequestHandler<Map<String, Object>,
 Map<String, Object>> {

    @SuppressWarnings("unchecked")
    @Override
    public Map<String, Object> handleRequest(Map<String, Object> event, Context
 context) {
        List<Map<String, Object>> failures = new ArrayList<>();
        Map<String, List<Map<String, Object>>> records =
                (Map<String, List<Map<String, Object>>>) event.getOrDefault("records",
 Map.of());

        for (Map.Entry<String, List<Map<String, Object>>> entry : records.entrySet()) {
            for (Map<String, Object> record : entry.getValue()) {
                String topic = (String) record.get("topic");
                Object partition = record.get("partition");
                Object offset = record.get("offset");
                String valueBase64 = (String) record.get("value");

                try {
                    String data = new String(Base64.getDecoder().decode(valueBase64),
 "UTF-8");
```

```
                    processMessage(data);
                } catch (Exception e) {
                    System.err.printf("Failed to process record topic=%s partition=%s
 offset=%s: %s%n",
                            topic, partition, offset, e.getMessage());
                    Map<String, Object> itemIdentifier = new HashMap<>();
                    itemIdentifier.put("partition", topic + "-" + partition);
                    itemIdentifier.put("offset", offset instanceof Number ? ((Number)
 offset).longValue() : null);
                    Map<String, Object> failure = new HashMap<>();
                    failure.put("itemIdentifier", itemIdentifier);
                    failures.add(failure);
                }
            }
        }

        Map<String, Object> response = new HashMap<>();
        response.put("batchItemFailures", failures);
        return response;
    }

    private void processMessage(String data) {
        // Your business logic for a single message
    }
}
```

Erfassen verworfener Batches für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen

Um Aufzeichnungen über fehlgeschlagene Aufrufe der Ereignisquellenzuordnung beizubehalten, fügen Sie der Ereignisquellenzuordnung Ihrer Ereignisfunktion ein Ziel hinzu. Jeder an das Ziel gesendete Datensatz ist ein JSON-Dokument mit Metadaten über den fehlgeschlagenen Aufruf. Bei Amazon S3-Zielen sendet Lambda auch den gesamten Aufrufdatensatz zusammen mit den Metadaten. Sie können jedes Amazon SNS SNS-Thema, jede Amazon SQS SQS-Warteschlange, jeden Amazon S3 S3-Bucket oder Kafka als Ziel konfigurieren.

Bei Amazon-S3-Zielen können Sie das Feature Amazon S3 Event Notifications verwenden, um Benachrichtigungen zu erhalten, wenn Objekte in Ihren S3-Ziel-Bucket hochgeladen werden. Sie können S3-Ereignisbenachrichtigungen auch so konfigurieren, dass eine weitere Lambda-Ereignisfunktion aufgerufen wird, um die automatische Verarbeitung fehlgeschlagener Batches durchzuführen.

Ihre Ausführungsrolle muss über Berechtigungen für das Ziel verfügen:

- Für ein SQS-Ziel: sqs: SendMessage
- Für ein SNS-Ziel: sns:Publish
- Für ein S3-Ziel: s3: und s3: PutObject ListBucket
- Für ein Kafka-Ziel: kafka-cluster: WriteData

Sie können ein Kafka-Thema als Ziel bei einem Ausfall für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das angegebene Kafka-Thema. Weitere Informationen finden Sie unter the section called “Kafka-Ziel bei einem Ausfall”.

Sie müssen einen VPC-Endpunkt für Ihren Fehlerziel-Service innerhalb Ihrer Kafka-Cluster-VPC bereitstellen.

Wenn Sie einen KMS-Schlüssel für Ihr Ziel konfiguriert haben, benötigt Lambda außerdem je nach Zieltyp folgende Berechtigungen:

- Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für ein S3-Ziel aktiviert haben, ist kms: erforderlich. GenerateDataKey Wenn sich der KMS-Schlüssel und das S3-Bucket-Ziel in einem anderen Konto befinden als Ihre Lambda-Ereignisfunktion und Ausführungsrolle,

konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, um kms: GenerateDataKey zuzulassen.

- Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für das SQS-Ziel aktiviert haben, sind kms:Decrypt und kms: erforderlich. GenerateDataKey Wenn sich der KMS-Schlüssel und das SQS-Warteschlangenziel in einem anderen Konto als Ihre Lambda-Ereignisfunktion und Ausführungsrolle befinden, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut und kms:Decrypt, kms:GenerateDataKey, kms: und kms: zulässt. DescribeKey ReEncrypt
- Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für das SNS-Ziel aktiviert haben, sind kms:Decrypt und kms: erforderlich. GenerateDataKey Wenn sich der KMS-Schlüssel und das SNS-Themenziel in einem anderen Konto befinden als Ihre Lambda-Ereignisfunktion und Ausführungsrolle, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut und kms:Decrypt, kms:GenerateDataKey, kms: und kms: zulässt. DescribeKey ReEncrypt

Konfigurieren von Fehlerzielen für eine Kafka-Zuordnung von Ereignisquellen

Gehen Sie folgendermaßen vor, um ein Ausfallziel mit der Konsole zu konfigurieren:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie als Quelle die Option Aufruf der Zuordnung von Ereignisquellen aus.
5. Wählen Sie für die Zuordnung von Ereignisquellen eine Ereignisquelle aus, die für diese Funktion konfiguriert ist.
6. Wählen Sie für Bedingung die Option Bei Ausfall aus. Für Aufrufe zur Zuordnung von Ereignisquellen ist dies die einzig akzeptierte Bedingung.
7. Wählen Sie unter Zieltyp den Zieltyp aus, an den Lambda Aufrufdatensätze sendet.
8. Wählen Sie unter Destination (Ziel) eine Ressource aus.
9. Wählen Sie Speichern.

Sie können mit der AWS CLI auch ein Ausfallziel konfigurieren. Mit dem folgenden Befehl create-event-source-mapping wird beispielsweise eine Zuordnung von Ereignisquellen mit einem SQS-Ziel zu `MyFunction` hinzugefügt:

```
aws lambda create-event-source-mapping \
```

```
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

Der folgende Befehl update-event-source-mapping fügt der verknüpften Ereignisquelle ein S3-Ausfallziel mit der Eingabe `uuid` hinzu:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

Um ein Ziel zu entfernen, geben Sie eine leere Zeichenfolge als Argument für den `destination-config`-Parameter an:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Bewährte Methoden für die Sicherheit in Amazon S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze aus Ihrer Event-Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Lambda-Ereignisfunktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:

    a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.

b. Wählen Sie im Richtlinien-Editor JSON aus.

c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.

d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.

e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle (CLI) einer Event-Funktion eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

SNS- und SQS-Beispiel-Aufrufsatz

Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kafka-Ereignisquelle an ein SNS-Thema oder eine SQS-Warteschlange sendet. Jeder der Schlüssel unter `recordsInfo` enthält sowohl das Kafka-Thema als auch die Kafka-Partition, getrennt durch einen Bindestrich. Bei dem Schlüssel `"Topic-0"` handelt es sich beispielsweise bei `Topic` um das Kafka-Thema und bei `0` um die Partition. Für jedes Thema und jede Partition können Sie die Offsets und Zeitstempeldaten verwenden, um die ursprünglichen Aufrufdatensätze zu finden.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
```

```
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3-Ziel Beispiel-Aufrufsatz

Für S3-Ziele sendet Lambda den gesamten Aufrufdatensatz zusammen mit den Metadaten an das Ziel. Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kafka-Ereignisquelle an ein S3-Bucket-Ziel sendet. Zusätzlich zu allen Feldern aus dem vorherigen Beispiel für SQS- und SNS-Ziele enthält das Feld `payload` den ursprünglichen Aufrufdatensatz als maskierte JSON-Zeichenfolge.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> Wir empfehlen außerdem, die S3-Versionsverwaltung in Ihrem Ziel-Bucket zu aktivieren.

Ein Kafka-Thema als Ziel für den Fall eines Fehlers verwenden

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das angegebene Kafka-Thema. Wenn Sie sowohl unendliche Wiederholungsversuche als auch ein Ziel für den Fall eines Fehlers konfigurieren, wendet Lambda automatisch maximal 10 Wiederholungsversuche an.

So funktioniert ein Kafka-Ziel für den Fall eines Fehlers

Wenn Sie ein Kafka-Thema als Ziel für den Fall eines Fehlers konfigurieren, fungiert Lambda als Kafka-Producer und schreibt fehlgeschlagene Datensätze in das Zielthema. Auf diese Weise entsteht in Ihrer Kafka-Infrastruktur ein Muster mit veralteten Themen (DLT).

- Gleiche Clusteranforderung — Das Zielthema muss im selben Kafka-Cluster wie Ihre Quellthemen vorhanden sein.
- Tatsächlicher Datensatzinhalt — Kafka-Ziele erhalten die tatsächlich fehlgeschlagenen Datensätze zusammen mit den Fehlermetadaten.
- Rekursionsprävention — Lambda verhindert Endlosschleifen, indem es Konfigurationen blockiert, bei denen Quell- und Zielthemen identisch sind.

Authentifizierung für ein Kafka-Ziel

Ein Kafka-Ziel bei einem Ausfall hat keine eigenen Authentifizierungseinstellungen. Das Zielthema befindet sich im selben Cluster wie Ihre Quellthemen, sodass Lambda fehlgeschlagene Datensätze mit der Authentifizierung schreibt, die Sie bereits im `SourceAccessConfigurations` Parameter der Ereignisquellenzuordnung konfiguriert haben.

Dies gilt für alle Authentifizierungsmethoden, die Lambda für Kafka unterstützt, einschließlich OAuth 2.0 (`OAUTHBEARER_AUTH`), IAM () und IAM `IAM_AUTH` with (). SASL/OAUTHBEARER `IAM_OAUTHBEARER_AUTH` Wenn Sie sich mit OAuth 2.0 authentifizieren, verwendet Lambda denselben Identitätsanbieter und dasselbe Geheimnis, um ein Token für die Erstellung für das Zielthema abzurufen. Konfigurieren Sie die Authentifizierung einmal in der Konfiguration für den Quellzugriff. Weitere Informationen finden Sie unter the section called “Cluster-Authentifizierung” Für selbstverwaltetes Kafka und the section called “Cluster-Authentifizierung” für Amazon MSK.

Stellen Sie sicher, dass die Identität, mit der Sie sich authentifizieren, in das Zielthema schreiben kann und nicht nur aus Ihren Quellthemen lesen kann. Bei der IAM-Authentifizierung bedeutet `kafka-cluster:WriteData` das, dass es sich um das Zielthema handelt. Bei anderen Methoden bedeutet das die entsprechende Schreibberechtigung in den Autorisierungsregeln Ihres Clusters.

Konfiguration eines Kafka-Ziels für den Fall eines Fehlers

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers konfigurieren, wenn Sie eine Kafka-Ereignisquellenzuordnung erstellen oder aktualisieren.

Konfiguration eines Kafka-Ziels (Konsole)

Um ein Kafka-Thema als Ziel für den Fall eines Fehlers zu konfigurieren (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Führen Sie eine der folgenden Aktionen aus:
   - Um einen neuen Kafka-Trigger hinzuzufügen, wählen Sie unter Funktionsübersicht die Option Trigger hinzufügen aus.
   - Um einen vorhandenen Kafka-Trigger zu ändern, wählen Sie den Auslöser aus und klicken Sie dann auf Bearbeiten.
4. Wählen Sie unter Zusätzliche Einstellungen als On-failure Ziel die Option Kafka-Thema aus.
5. Geben Sie als Themenname den Namen des Kafka-Themas ein, an das Sie fehlgeschlagene Datensätze senden möchten.
6. Wählen Sie Hinzufügen oder Speichern.

Konfiguration eines Kafka-Ziels (AWS CLI)

Verwenden Sie das `kafka://` Präfix, um ein Kafka-Thema als Ziel für den Fall eines Fehlers anzugeben.

Erstellen einer Ereignisquellenzuordnung mit einem Kafka-Ziel

Im folgenden Beispiel wird eine Amazon MSK-Ereignisquellenzuordnung mit einem Kafka-Thema als Ziel für den Fall eines Fehlers erstellt:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
```

```
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Verwenden Sie für selbstverwaltetes Kafka dieselbe Syntax:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Aktualisierung einer Kafka-Destination

Verwenden Sie den `update-event-source-mapping` Befehl, um ein Kafka-Ziel hinzuzufügen oder zu ändern:

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Datensatzformat für ein Kafka-Ziel

Wenn Lambda fehlgeschlagene Datensätze an ein Kafka-Thema sendet, enthält jede Nachricht sowohl Metadaten über den Fehler als auch den eigentlichen Datensatzinhalt.

Metadaten für den Fehler

Die Metadaten enthalten Informationen darüber, warum der Datensatz fehlgeschlagen ist, und Details zum ursprünglichen Batch:

```
{
  "requestContext": {
    "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "condition": "RetriesExhausted",
```

```
    "approximateInvokeCount": 3
  },
  "responseContext": {
    "statusCode": 200,
    "executedVersion": "$LATEST",
    "functionError": "Unhandled"
  },
  "version": "1.0",
  "timestamp": "2019-11-14T18:16:05.568Z",
  "KafkaBatchInfo": {
    "batchSize": 1,
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "payloadSize": 1162,
    "recordInfo": {
      "offset": "49601189658422359378836298521827638475320189012309704722",
      "timestamp": "2019-11-14T18:16:04.835Z"
    }
  },
  "payload": {
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "records": {
      "my-topic-0": [
        {
          "headers": [],
          "key": "dGVzdC1rZXk=",
          "offset": 100,
          "partition": 0,
          "timestamp": 1749116692330,
          "timestampType": "CREATE_TIME",
          "topic": "my-topic",
          "value": "dGVzdC12YWx1ZQ=="
        }
      ]
    }
  }
}
```

Verhalten der Partitionsschlüssel

Lambda verwendet beim Produzieren zum Zielthema denselben Partitionsschlüssel wie im Originaldatensatz. Wenn der ursprüngliche Datensatz keinen Schlüssel hatte, verwendet Lambda die standardmäßige Round-Robin-Partitionierung von Kafka für alle verfügbaren Partitionen im Zielthema.

Anforderungen und Einschränkungen

- Bereitgestellter Modus erforderlich — Ein Kafka-Ziel für den Fall eines Fehlers ist nur für Zuordnungen von Ereignisquellen verfügbar, bei denen der Bereitstellungsmodus aktiviert ist.
- Nur derselbe Cluster — Das Zielthema muss im selben Kafka-Cluster wie Ihre Quellthemen vorhanden sein.
- Themenberechtigungen — Ihre Zuordnung zur Ereignisquelle muss über Schreibberechtigungen für das Zielthema verfügen. Beispiel:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ClusterPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeCluster",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
                "arn:aws:kafka:*:*:cluster/*"
            ]
        },
        {
            "Sid": "TopicPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
```

```
                "arn:aws:kafka:*:*:topic/*/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kafka:DescribeCluster",
                "kafka:GetBootstrapBrokers",
                "kafka:Produce"
            ],
            "Resource": "arn:aws:kafka:*:*:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

- Keine Rekursion — Der Name des Zielthemas darf nicht mit einem Ihrer Quellthemennamen identisch sein.

Protokollierung der Zuordnung von Kafka-Ereignisquellen

Sie können die Protokollierung auf Systemebene für Ihre Kafka-Ereignisquellenzuordnungen konfigurieren, um die Systemprotokolle zu aktivieren und zu filtern, an die Lambda-Ereignisabfragen senden. CloudWatch

Diese Funktion ist nur für Kafka-Ereignisquellenzuordnungen und im Bereitstellungsmodus verfügbar.

Für die Zuordnung von Ereignisquellen mit Protokollierungskonfiguration können Sie jetzt auch die Systemprotokolle aus vorgefertigten Protokollabfragen auf der Registerkarte Monitor auf der Seite Konsole Lambda > Zusätzliche Ressourcen > Ereignisquellenzuordnungen überprüfen.

Wie funktioniert die Protokollierung

Wenn Sie die Protokollierungskonfiguration mit Protokollebene in Ihrer Ereignisquellenzuordnung festlegen, sendet der Lambda-Ereignisabfrage entsprechende Protokolle (Systemprotokolle für die Ereignisquellenzuweisung).

Die Zuordnung der Ereignisquelle verwendet dasselbe Protokollziel wie Ihre Lambda-Ereignisfunktion erneut. Stellen Sie sicher, dass die Ausführungsrolle Ihrer Lambda-Ereignisfunktion über die erforderlichen Protokollierungsberechtigungen verfügt.

Die Zuordnung der Ereignisquelle wird über einen eigenen Protokollstream verfügen, mit Datum und UUID für die Zuordnung der Ereignisquelle als Protokollstreamname, z. B. `2020/01/01/12345678-1234-1234-1234-12345678901`

Für Systemprotokolle zur Zuordnung von Ereignisquellen können Sie zwischen den folgenden Protokollebenen wählen.

Protokollebene	Usage
DEBUG (am detailliertesten)	Detaillierte Informationen zum Fortschritt der Verarbeitung der Ereignisquelle
INFO	Meldungen über den normalen Betrieb Ihrer Ereignisquellenzuordnung
WARN (am wenigsten Details)	Meldungen über mögliche Warnungen und Fehler, die zu unerwartetem Verhalten führen könnten

Wenn Sie eine Protokollebene auswählen, sendet der Lambda-Ereignisabfrage Protokolle auf dieser Ebene und niedriger. Wenn Sie beispielsweise die Systemprotokollebene für die Zuordnung von Ereignisquellen auf INFO setzen, sendet der Event-Poller keine Protokollausgaben auf der DEBUG-Ebene.

Konfigurieren der Protokollierung

Sie können die Protokollierungskonfiguration festlegen, wenn Sie eine Kafka-Ereignisquellenzuordnung erstellen oder aktualisieren.

Konfiguration der Protokollierung (Konsole)

Um die Protokollierung zu konfigurieren (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Führen Sie eine der folgenden Aktionen aus:
   - Um einen neuen Kafka-Trigger hinzuzufügen, wählen Sie unter Funktionsübersicht die Option Trigger hinzufügen aus.
   - Um einen vorhandenen Kafka-Trigger zu ändern, wählen Sie den Auslöser aus und klicken Sie dann auf Bearbeiten.
4. Aktivieren Sie unter Konfiguration des Event-Pollers für den Bereitstellungsmodus das Kontrollkästchen Konfigurieren. Und die Einstellung für die Protokollebene würde angezeigt.
5. Wählen Sie die Dropdownliste Protokollebene und wählen Sie eine Ebene für die Zuordnung der Ereignisquellen aus.
6. Wählen Sie unten Hinzufügen oder Speichern, um die Zuordnung der Ereignisquellen zu erstellen oder zu aktualisieren.

Protokollierung konfigurieren (AWS CLI)

Erstellen einer Ereignisquellenzuordnung mit Protokollierung

Das folgende Beispiel erstellt eine Amazon MSK-Ereignisquellenzuordnung mit Protokollierungskonfiguration:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
```

```
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --logging-config '{"SystemLogLevel":"DEBUG"}'
```

Verwenden Sie für selbstverwaltetes Kafka dieselbe Syntax:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --logging-config '{"SystemLogLevel":"DEBUG"}'
```

Die Logging-Konfiguration wird aktualisiert

Verwenden Sie den `update-event-source-mapping` Befehl, um die Protokollierungskonfiguration hinzuzufügen oder zu ändern:

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --logging-config '{"SystemLogLevel":"WARN"}'
```

Aufzeichnungsformat für ein Systemprotokoll zur Zuordnung von Kafka-Ereignisquellen

Wenn der Lambda-Ereignisabfrage das Protokoll sendet, enthält jeder Protokolleintrag allgemeine Metadaten zur Zuordnung von Ereignisquellen sowie ereignisspezifische Inhalte.

WARN-Protokolleintrag

Der WARN-Datensatz enthält Fehler oder Warnungen vom Event-Poller und wird ausgegeben, als das Ereignis eingetreten ist. Beispiel:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1546347650000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
```

```
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Timeout expired while fetching topic metadata",
        "errorCode": "org.apache.kafka.common.errors.TimeoutException"
    }
}
```

INFO-Protokolleintrag

Der INFO-Datensatz enthält Kafka-Consumer-Client-Konfigurationen in jedem Event-Poller und wird ausgegeben, wenn ein Consumer erstellt oder geändert wird. Beispiel:

```
{
    "eventType": "POLLER_STATUS_EVENT",
    "timestamp": 1546347660000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "INFO",
    "kafkaEventSourceConnection": {
        "brokerEndpoints": "boot-abcd1234.c2.kafka-serverless.us-
east-1.amazonaws.com:9098",
        "consumerId": "12345678-1234-1234-1234-123456789012-0",
        "topics": [
            "test"
        ],
        "consumerGroupId": "12345678-1234-1234-1234-123456789012",
        "securityProtocol": "SASL_SSL",
        "saslMechanism": "AWS_MSK_IAM",
        "totalPartitionCount": 2,
        "assignedPartitionCount": 2,
        "partitionsAssignmentGeneration": 5,
        "assignedPartitions": [
            "test-0",
            "test-1"
        ],
        "networkConfig": {
            "ipAddresses": [
```

```
                "10.100.141.1"
            ],
            "subnetCidrBlock": "10.100.128.0/20",
            "securityGroups": [
                "sg-abcdefabcdefabcdef"
            ]
        }
    }
}
```

DEBUG-Protokolleintrag

Das DEBUG-Protokoll enthält die Informationen zu den Kafka-Offsets bei der Verarbeitung der Ereignisquellenzuordnung, und die Offsetinformationen werden pro Minute ausgegeben. Beispiel:

```
{
    "eventType": "KAFKA_STATUS_EVENT",
    "timestamp": 1546347670000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "DEBUG",
    "kafkaPartitionOffsets": {
        "partition": "test-1",
        "endOffset": 5004,
        "consumedOffset": 5003,
        "processedOffset": 5003,
        "committedOffset": 5004
    }
}
```

Fehlerbehebung bei der Kafka-Zuordnung von Ereignisquellen

Die folgenden Themen enthalten Hinweise zur Fehlerbehebung für Fehler und Probleme, die bei der Verwendung von Amazon MSK oder selbstverwaltetem Apache Kafka mit Lambda auftreten können.

Weitere Hilfe zur Fehlerbehebung finden Sie im AWS -Wissenszentrum.

Authentifizierungs- und Autorisierungsfehler

Wenn eine der für die Nutzung von Daten aus dem Kafka-Cluster erforderlichen Berechtigungen fehlt, zeigt Lambda in der Ereignisquellenzuordnung unter eine der folgenden Fehlermeldungen an. LastProcessingResult

Fehlermeldungen

- Cluster konnte Lambda nicht autorisieren
- SASL-Authentifizierung fehlgeschlagen
- Server konnte Lambda nicht authentifizieren
- Lambda konnte Server nicht authentifizieren
- Bereitgestelltes Zertifikat oder bereitgestellter privater Schlüssel ist ungültig

Cluster konnte Lambda nicht autorisieren

Bei SASL/SCRAM oder mTLS weist dieser Fehler darauf hin, dass der angegebene Benutzer nicht über alle der folgenden erforderlichen Kafka-Zugriffssteuerungslisten (ACL) -Berechtigungen verfügt:

- DescribeConfigs Cluster
- Beschreiben von Gruppe
- Gruppe lesen
- Thema beschreiben
- Thema lesen

Wenn Sie Kafka-ACLs mit den erforderlichen `kafka-cluster`-Berechtigungen erstellen, geben Sie das Thema und die Gruppe als Ressourcen an. Der Themenname muss mit dem Thema in der Ereignisquellenzuordnung übereinstimmen. Der Gruppenname muss mit der UUID der Ereignisquellenzuordnung übereinstimmen.

Nachdem Sie der Ausführungsrolle die erforderlichen Berechtigungen hinzugefügt haben, kann es einige Minuten dauern, bis die Änderungen wirksam werden.

Im Folgenden finden Sie ein Beispiel für ein ESM-Protokoll auf Systemebene, nachdem Logging Config für dieses Problem aktiviert wurde:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567890123,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE11111-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Not authorized to access topics: [my-topic]",
        "errorCode": "org.apache.kafka.common.errors.TopicAuthorizationException"
    }
}
```

SASL-Authentifizierung fehlgeschlagen

Bei SASL/SCRAM oder SASL/PLAIN weist dieser Fehler darauf hin, dass die angegebenen Anmeldeinformationen nicht gültig sind.

Zur IAM-Zugriffskontrolle fehlt der Ausführungsrolle die Berechtigung `kafka-cluster:Connect` für den Cluster. Fügen Sie der Rolle diese Berechtigung hinzu und geben Sie den Amazon-Ressourcenname (ARN) des Clusters als Ressource an.

Dieser Fehler wird Ihnen möglicherweise in zeitlichen Abständen angezeigt. Der Cluster lehnt Verbindungen ab, wenn die Anzahl der TCP-Verbindungen das Servicekontingent überschreitet. Lambda zieht sich zurück und versucht es erneut, bis eine Verbindung erfolgreich ist. Nachdem Lambda eine Verbindung zum Cluster hergestellt hat und nach Datensätzen abfragt, ändert sich das letzte Verarbeitungsergebnis zu `OK`.

Im Folgenden finden Sie ein Beispiel für ein ESM-Protokoll auf Systemebene nach der Aktivierung von Logging Config für dieses Problem bei Verwendung der IAM-Authentifizierung:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567890456,
```

```
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE22222-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "[a1b2c3d4-5678-90ab-cdef-EXAMPLE22222]: Access denied",
        "errorCode": "org.apache.kafka.common.errors.SaslAuthenticationException"
    }
}
```

Server konnte Lambda nicht authentifizieren

Dieser Fehler weist darauf hin, dass der Kafka-Broker Lambda nicht authentifizieren konnte. Dieser Fehler kann aus folgenden Gründen auftreten:

- Sie haben kein Client-Zertifikat für die mTLS-Authentifizierung bereitgestellt.
- Sie haben ein Client-Zertifikat bereitgestellt, aber die Kafka-Broker sind nicht für die Verwendung der mTLS-Authentifizierung konfiguriert.
- Die Kafka-Broker vertrauen einem Client-Zertifikat nicht.

Lambda konnte Server nicht authentifizieren

Dieser Fehler weist darauf hin, dass Lambda den Kafka-Broker nicht authentifizieren konnte. Dieser Fehler kann aus folgenden Gründen auftreten:

- Für selbstverwaltetes Apache Kafka: Die Kafka-Broker verwenden selbstsignierte Zertifikate oder eine private Zertifizierungsstelle (CA), haben jedoch das Root-CA-Zertifikat des Servers nicht bereitgestellt.
- Für selbstverwaltetes Apache Kafka: Das Root-CA-Zertifikat des Servers stimmt nicht mit der Root-CA überein, die das Zertifikat des Brokers signiert hat.
- Die Überprüfung des Hostnamens ist fehlgeschlagen, weil das Zertifikat des Brokers nicht den DNS-Namen oder die IP-Adresse des Brokers als alternativen Betreffnamen enthält.

Bereitgestelltes Zertifikat oder bereitgestellter privater Schlüssel ist ungültig

Dieser Fehler weist darauf hin, dass der Kafka-Konsument das bereitgestellte Zertifikat oder den bereitgestellten privaten Schlüssel nicht verwenden konnte. Stellen Sie sicher, dass das Zertifikat

und der Schlüssel das PEM-Format haben und die Verschlüsselung des privaten Schlüssels einen PBES1-Algorithmus nutzt.

Im Folgenden finden Sie ein Beispiel für ein ESM-Protokoll auf Systemebene, nachdem Logging Config für dieses Problem aktiviert wurde:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567891234,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE44444",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE44444-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE44444/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Invalid PEM keystore configs",
        "errorCode": "org.apache.kafka.common.errors.InvalidConfigurationException"
    }
}
```

Netzwerk- und Verbindungsfehler

Probleme mit der Netzwerkkonfiguration können verhindern, dass Lambda eine Verbindung zu Ihrem Kafka-Cluster herstellt. In den folgenden Themen werden häufig auftretende Netzwerkfehler beschrieben.

Fehlermeldungen

- Verbindungstimeout aufgrund der Konfiguration der Sicherheitsgruppe
- Kafka-Broker-Endpunkte können nicht aufgelöst werden

Verbindungstimeout aufgrund der Konfiguration der Sicherheitsgruppe

Wenn die mit Ihrem Kafka-Cluster verknüpfte Sicherheitsgruppe keinen eingehenden Datenverkehr von sich selbst zulässt, kann Lambda keine Verbindung zum Cluster herstellen. Stellen Sie sicher, dass die Regeln der Sicherheitsgruppe für eingehenden Datenverkehr von der Sicherheitsgruppe selbst auf den Kafka-Broker-Ports zulassen.

Im Folgenden finden Sie ein Beispiel für ein ESM-Protokoll auf Systemebene, nachdem Logging Config für dieses Problem aktiviert wurde:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567892345,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE55555",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE55555-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE55555/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Timeout expired while fetching topic metadata",
        "errorCode": "org.apache.kafka.common.errors.TimeoutException"
    }
}
```

Sie können auch das Kafka-Consumer-INFO-Protokoll überprüfen, um die Verbindung und die Netzwerkkonfiguration zu überprüfen. Das `brokerEndpoints` Feld zeigt die Adressen des Kafka-Brokers `securityProtocol` und `saslMechanism` (falls zutreffend) die Authentifizierungsmethode an, und das `networkConfig` Feld zeigt die IP-Adressen, den Subnetz-CIDR-Block und die Sicherheitsgruppen, die für die Zuordnung der Ereignisquellen verwendet wurden. Stellen Sie sicher, dass die aufgelisteten Sicherheitsgruppen den erforderlichen eingehenden Datenverkehr zulassen:

```
{
    "eventType": "POLLER_STATUS_EVENT",
    "timestamp": 1734567892456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-11111EXAMPLE",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE/0",
    "logLevel": "INFO",
    "kafkaEventSourceConnection": {
        "brokerEndpoints": "boot-abcd1234.c2.kafka-serverless.us-
east-1.amazonaws.com:9098",
        "consumerId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE-0",
        "topics": [
            "my-topic"
        ],
        "consumerGroupId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE",
        "securityProtocol": "SASL_SSL",
        "saslMechanism": "AWS_MSK_IAM",
        "totalPartitionCount": 2,
```

```
        "assignedPartitionCount": 2,
        "partitionsAssignmentGeneration": 1,
        "assignedPartitions": [
            "my-topic-0",
            "my-topic-1"
        ],
        "networkConfig": {
            "ipAddresses": [
                "10.0.0.37"
            ],
            "subnetCidrBlock": "10.0.0.32/28",
            "securityGroups": [
                "sg-0123456789abcdef0"
            ]
        }
    }
}
```

Kafka-Broker-Endpunkte können nicht aufgelöst werden

Dieser Fehler weist darauf hin, dass der Kafka-Cluster nicht existiert oder gelöscht wurde. Stellen Sie sicher, dass der in der Ereignisquellenzuordnung angegebene Cluster vorhanden ist und sich in einem aktiven Zustand befindet.

Im Folgenden finden Sie ein Beispiel für ein ESM-Protokoll auf Systemebene, nachdem Logging Config für dieses Problem aktiviert wurde:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567893456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE66666",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE66666-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE66666/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "No resolvable bootstrap urls given in bootstrap.servers",
        "errorCode": "org.apache.kafka.common.config.ConfigException"
    }
}
```

Fehler bei der Ereignisquellen-Zuweisung

Wenn Sie Ihren Apache Kafka-Cluster als Ereignisquelle für Ihre Lambda-Ereignisfunktion hinzufügen und Ihre Funktion auf einen Fehler stößt, stoppt Ihr Kafka-Consumer die Verarbeitung von Datensätzen. Verbraucher einer Themenpartition sind diejenigen, die Ihre Datensätze abonnieren, lesen und verarbeiten. Ihre anderen Kafka-Verbraucher können weiterhin Datensätze verarbeiten, sofern sie nicht auf denselben Fehler stoßen.

Um die Ursache eines gestoppten Verbrauchers zu ermitteln, überprüfen Sie das `StateTransitionReason`-Feld in der Antwort von `EventSourceMapping`. In der folgenden Liste werden die Ereignisquellfehler beschrieben, die Sie erhalten können:

`ESM_CONFIG_NOT_VALID`

Die Konfiguration der Ereignisquellenzuordnung ist ungültig.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda konnte die Ereignisquelle nicht authentifizieren.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda verfügt nicht über die erforderlichen Berechtigungen für den Zugriff auf die Ereignisquelle.

`FUNCTION_CONFIG_NOT_VALID`

Die Konfiguration der Funktion ist ungültig.

> **Note**
>
> Wenn Ihre Lambda-Ereignisdatensätze die zulässige Größenbeschränkung von 6 MB überschreiten, können sie unbearbeitet bleiben.

Aufrufen einer Lambda-Ereignisfunktion mithilfe eines Amazon API Gateway Gateway-Endpunkts

Mithilfe von Amazon API Gateway können Sie eine Web-API mit einem HTTP-Endpunkt für Ihre Lambda Event Function erstellen. API Gateway bietet Tools zum Erstellen und Dokumentieren von Web-APIs, die HTTP-Anfragen an Lambda Event Functions weiterleiten. Sie können den Zugriff auf Ihre API mit Authentifizierungs- und Autorisierungskontrollen sichern. Ihre APIs können Datenverkehr über das Internet bereitstellen oder nur innerhalb Ihrer VPC zugänglich sein.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: API Gateway und Lambda-Funktions-URLs. Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “API Gateway im Vergleich zu Funktions-URLs”.

Ressourcen in Ihrer API definieren mindestens eine Methode, z. B. GET oder POST. Methoden haben eine Integration, die Anfragen an eine Lambda-Ereignisfunktion oder einen anderen Integrationstyp weiterleitet. Sie können jede Ressource und jede Methode einzeln definieren oder spezielle Ressourcen- und Methodenarten verwenden, um alle Anforderungen abzugleichen, die einem Muster entsprechen. Eine Proxy-Ressource fängt alle Pfade unter einer Ressource ab. Die ANY-Methode fängt alle HTTP-Methoden ab.

Sections

- Auswählen eines API-Typs
- Hinzufügen eines Endpunkts zu Ihrer Lambda-Ereignisfunktion
- Proxy-Integration
- Ereignisformat
- Reaktionsformat
- Berechtigungen
- Beispielanwendung
- Der Event-Handler von Powertools für AWS Lambda
- Tutorial: Verwenden von Lambda mit API Gateway

- Behandlung von Lambda-Fehlern mit einer API-Gateway-API
- Wählen Sie eine Methode aus, um Ihre Lambda-Ereignisfunktion mithilfe einer HTTP-Anfrage aufzurufen

Auswählen eines API-Typs

API Gateway unterstützt drei Arten von APIs, die Lambda Event Functions aufrufen:

- HTTP API: Eine leichte RESTful-API mit geringer Latenz.
- REST-API: Eine anpassbare, featurereiche RESTful-API.
- WebSocket API: Eine Web-API, die persistente Verbindungen mit Clients für die Vollduplex-Kommunikation unterhält.

HTTP-APIs und REST-APIs sind beides RESTful-APIs, die HTTP-Anforderungen verarbeiten und Antworten zurückgeben. HTTP-APIs sind neuer und werden mit der API der API-Gateway-Version 2 erstellt. Die folgenden Features sind neu bei HTTP-APIs:

HTTP-API-Features

- Automatische Bereitstellungen – wenn Sie Routen oder Integrationen ändern, werden Änderungen automatisch in Phasen bereitgestellt, bei denen die automatische Bereitstellung aktiviert ist.
- Standardphase – Sie können eine Standardphase (`$default`) erstellen, um Anforderungen am Stammpfad der URL Ihrer API zu bedienen. Bei benannten Phasen müssen Sie den Schrittnamen am Anfang des Pfades angeben.
- CORS-Konfiguration – Sie können Ihre API so konfigurieren, dass ausgehenden Antworten CORS-Header hinzugefügt werden, anstatt sie manuell in Ihrem Funktionscode hinzufügen zu müssen.

REST-APIs sind die klassischen RESTful-APIs, die von API Gateway seit dem Start unterstützt wurden. REST-APIs verfügen derzeit über mehr Anpassungs-, Integrations- und Verwaltungsfunktionen.

REST-API-Funktionen

- Integrationstypen – REST-APIs unterstützen benutzerdefinierte Lambda-Integrationen. Mit einer benutzerdefinierten Integration können Sie nur den Text der Anforderung an die Funktion senden oder eine Transformationsvorlage auf den Anforderungstext anwenden, bevor Sie sie an die Funktion senden.

- Zugriffskontrolle – REST-APIs unterstützen weitere Optionen für die Authentifizierung und Autorisierung.
- Überwachung und Ablaufverfolgung — REST-APIs unterstützen AWS X-Ray Tracing und zusätzliche Protokollierungsoptionen.

Einen detaillierten Vergleich finden Sie unter Auswählen zwischen HTTP-APIs und REST-APIs im API-Gateway-Entwicklerhandbuch.

WebSocket APIs verwenden auch die API Gateway Version 2 API und unterstützen einen ähnlichen Funktionsumfang. Verwenden Sie eine WebSocket API für Anwendungen, die von einer dauerhaften Verbindung zwischen dem Client und der API profitieren. WebSocket APIs bieten Vollduplex-Kommunikation, was bedeutet, dass sowohl der Client als auch die API kontinuierlich Nachrichten senden können, ohne auf eine Antwort warten zu müssen.

HTTP-APIs unterstützen ein vereinfachtes Ereignisformat (Version 2.0). Ein Beispiel für ein Ereignis aus einer HTTP-API finden Sie unter Erstellen von AWS Lambda Proxyintegrationen für HTTP-APIs in API Gateway.

Weitere Informationen finden Sie unter Erstellen von AWS Lambda Proxyintegrationen für HTTP-APIs in API Gateway.

Hinzufügen eines Endpunkts zu Ihrer Lambda-Ereignisfunktion

So fügen Sie Ihrer Lambda-Ereignisfunktion einen öffentlichen Endpunkt hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie API Gateway aus.
5. Wählen Sie Create an API (API erstellen) oder Use an existing API (Vorhandene API verwenden).

    a. Neue API: Wählen Sie als API type (API-Typ) HTTP API aus. Weitere Informationen finden Sie unter Auswählen eines API-Typs.

    b. Vorhandene API: Wählen Sie die API aus der Dropdown-Liste aus oder geben Sie die API-ID ein (z. B. r3pmxmplak).

6. Wählen Sie unter Security (Sicherheit) die Option Open (Öffnen) aus.
7. Wählen Sie Add aus.

Proxy-Integration

API-Gateway-APIs bestehen aus Phasen, Ressourcen, Methoden und Integrationen. Die Phase und die Ressource bestimmen den Pfad des Endpunkts:

API-Pfadformat

- `/prod/` – Die `prod`-Phase und die Root-Ressource.
- `/prod/user` – Die `prod`-Phase und die `user`-Ressource.
- `/dev/{proxy+}` – Jede Route in der `dev`-Phase.
- `/` – (HTTP-APIs) Die Standardphase und die Root-Ressource.

Eine Lambda-Integration ordnet einer Lambda-Event-Funktion eine Kombination aus Pfad und HTTP-Methode zu. Sie können API Gateway so konfigurieren, dass der Hauptteil der HTTP-Anforderung unverändert (benutzerdefinierte Integration) übergeben oder der Anforderungstext in einem Dokument mit allen Anforderungsinformationen, einschließlich Header, Ressource, Pfad und Methode, gekapselt wird.

Weitere Informationen finden Sie unter Lambda-Proxy-Integrationen in API Gateway.

Ereignisformat

Amazon API Gateway ruft Ihre Funktion synchron mit einem Ereignis auf, das eine JSON-Darstellung der HTTP-Anforderung enthält. Bei einer benutzerdefinierten Integration ist das Ereignis der Text der Anforderung. Bei einer Proxy-Integration hat das Ereignis eine definierte Struktur. Ein Beispiel für ein Proxy-Ereignis von einer API Gateway REST API finden Sie unter Eingabeformat einer Lambda-Funktion für die Proxy-Integration im API Gateway Developer Guide.

Reaktionsformat

API Gateway wartet auf eine Antwort von Ihrer Funktion und leitet das Ergebnis an den Aufrufer weiter. Für eine benutzerdefinierte Integration definieren Sie eine Integrationsantwort und eine Methodenantwort, um die Ausgabe von der Funktion in eine HTTP-Antwort zu konvertieren. Für eine Proxy-Integration muss die Funktion mit einer Darstellung der Antwort in einem bestimmten Format antworten.

Das folgende Beispiel zeigt ein Antwortobjekt aus einer Node.js Funktion. Das Antwortobjekt stellt eine erfolgreiche HTTP-Antwort dar, die ein JSON-Dokument enthält.

Example index.mjs — Antwortobjekt für die Proxyintegration () Node.js

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
      "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
    }
```

Die Lambda-Laufzeit serialisiert das Antwortobjekt in JSON und sendet es an die API. Die API analysiert die Antwort und verwendet sie zur Erstellung einer HTTP-Antwort verwendet, die sie dann an den Client sendet, der die ursprüngliche Anforderung gestellt hat.

Example HTTP-Antwort

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

Berechtigungen

Amazon API Gateway erhält die Berechtigung zum Aufrufen Ihrer Funktion über die ressourcenbasierte Richtlinie der Funktion. Sie können eine Aufrufberechtigung für eine gesamte

API erteilen oder einen eingeschränkten Zugriff auf eine Phase, eine Ressource oder eine Methode gewähren.

Wenn Sie Ihrer Funktion mithilfe der Lambda-Konsole, mithilfe der API-Gateway-Konsole oder in einer AWS SAM -Vorlage eine API hinzufügen, wird die ressourcenbasierte Richtlinie der Funktion automatisch aktualisiert. Es folgt eine Beispiel-Funktionsrichtlinie.

Example Funktionsrichtlinie

JSON

```
{
  "Version":"2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-
east-2:111122223333:ktyvxmpls1/*/GET/"
        }
      }
    }
  ]
}
```

Sie können Funktionsrichtlinienberechtigungen manuell mit den folgenden API-Operationen verwalten:

- AddPermission

- RemovePermission
- GetPolicy

Mit dem `add-permission`-Befehl können Sie einer vorhandenen API Aufrufberechtigung erteilen. Beispiel:

```
aws lambda add-permission \
  --function-name my-function \
  --statement-id apigateway-get --action lambda:InvokeFunction \
  --principal apigateway.amazonaws.com \
  --source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> Wenn sich Ihre Funktion und API unterscheiden AWS-Regionen, muss die Regionskennung im Quell-ARN mit der Region der Funktion übereinstimmen, nicht mit der Region der API. Bei Aufruf einer API-Gateway-Funktion verwendet diese einen Ressourcen-ARN, der auf dem ARN der API basiert, aber so geändert wird, dass er dem Bereich der Funktion entspricht.

Der Quell-ARN in diesem Beispiel erteilt eine Berechtigung für eine Integration in die GET-Methode der Root-Ressource in der Standardphase einer API mit ID `mnh1xmpli7`. Sie können ein Sternchen im Quell-ARN verwenden, um Berechtigungen für mehrere Phasen, Methoden oder Ressourcen zu erteilen.

Ressourcenmuster

- `mnh1xmpli7/*/GET/*` – GET-Methode bei allen Ressourcen in allen Phasen.

- `mnh1xmpli7/prod/ANY/user` – JEDE Methode bei der `user`-Ressource in der `prod`-Phase.
- `mnh1xmpli7/*/*/*` – Jede Methode bei allen Ressourcen in allen Phasen.

Weitere Informationen zum Anzeigen der Richtlinie und zum Entfernen von Anweisungen finden Sie unter Arbeiten mit ressourcenbasierten Richtlinien in Lambda.

Beispielanwendung

Das API Gateway mit Node.js Beispiel-App enthält eine Funktion mit einer AWS SAM Vorlage, die eine REST-API erstellt, für die die AWS X-Ray Ablaufverfolgung aktiviert ist. Es enthält auch Skripte zum Bereitstellen und Aufrufen der Funktion sowie zum Testen der API und zum Bereinigen.

Der Event-Handler von Powertools für AWS Lambda

Der Event-Handler aus dem Powertools for AWS Lambda-Toolkit bietet Routing, Middleware, CORS-Konfiguration, Generierung von OpenAPI-Spezifikationen, Anforderungsvalidierung, Fehlerbehandlung und andere nützliche Funktionen beim Schreiben von Lambda-Ereignisfunktionen, die von einem API-Gateway-Endpunkt (HTTP oder REST) aufgerufen werden. Das Event-Handler-Hilfsprogramm ist für Python und verfügbar TypeScript/JavaScript. Weitere Informationen finden Sie unter Event Handler REST API in der Powertools for AWS Lambda (Python) -Dokumentation und Event Handler HTTP API in der Powertools for AWS Lambda () -Dokumentation. TypeScript

Python

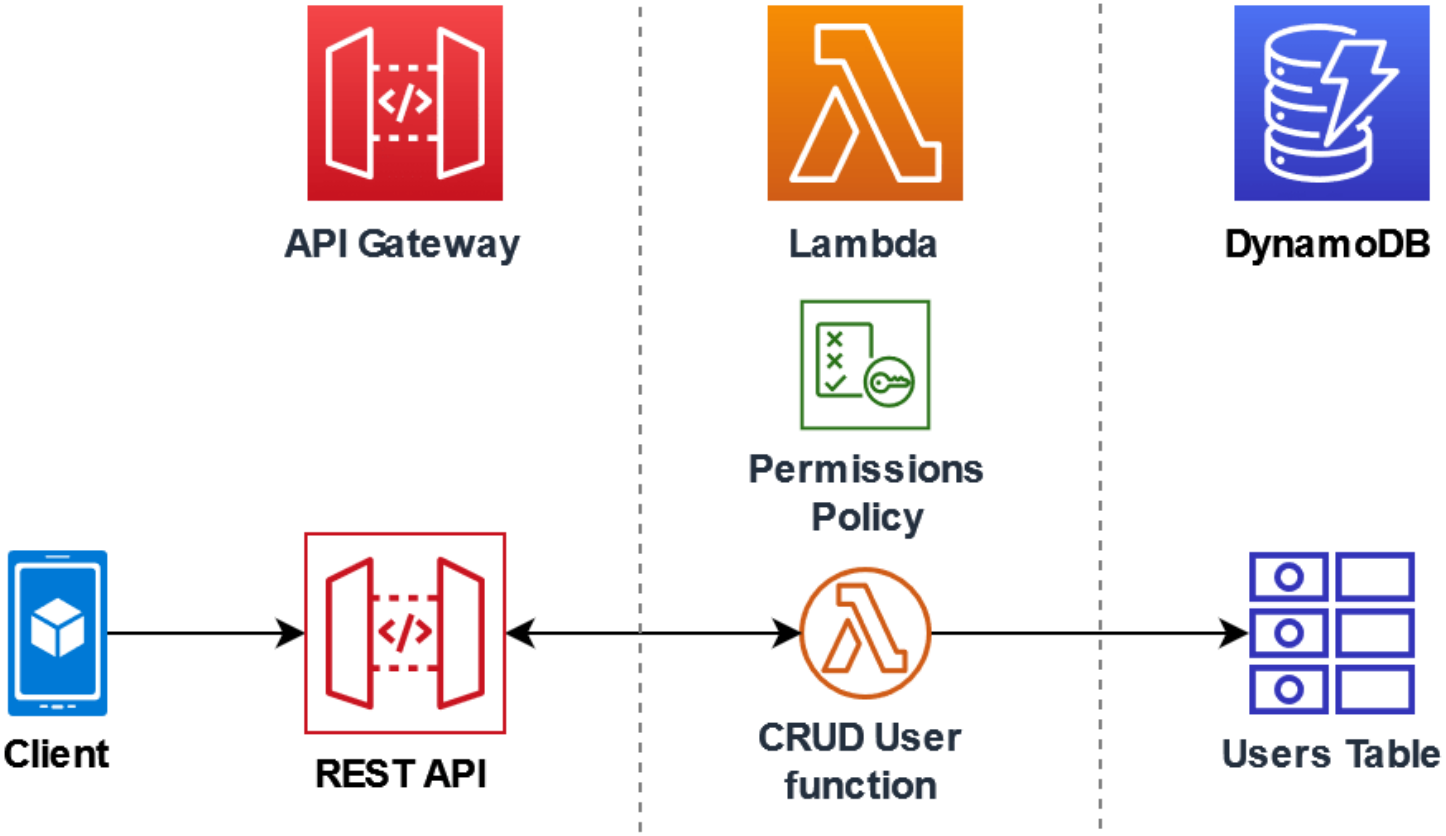
```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing.lambda_context import LambdaContext

app = APIGatewayRestResolver()
logger = Logger()

@app.get("/healthz")
def ping():
    return {"message": "health status ok"}

@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

TypeScript

```
import { Router } from '@aws-lambda-powertools/event-handler/experimental-rest';
import { Logger } from '@aws-lambda-powertools/logger';
import {
  correlationPaths,
  search,
} from '@aws-lambda-powertools/logger/correlationId';
import type { Context } from 'aws-lambda/handler';

const logger = new Logger({
  correlationIdSearchFn: search,
});

const app = new Router({ logger });

app.get("/healthz", async () => {
  return { message: "health status ok" };
});

export const handler = async (event: unknown, context: Context) => {
  // You can continue using other utilities just as before
  logger.addContext(context);
  logger.setCorrelationId(event, correlationPaths.API_GATEWAY_REST);
  return app.resolve(event, context);
};
```

Tutorial: Verwenden von Lambda mit API Gateway

In diesem Tutorial erstellen Sie eine REST-API, über die Sie mithilfe einer HTTP-Anfrage eine Lambda-Ereignisfunktion aufrufen. Ihre Lambda-Ereignisfunktion führt Erstellungs-, Lese-, Aktualisierungs- und Löschvorgänge (CRUD) in einer DynamoDB-Tabelle durch. Diese Funktion wird hier zur Veranschaulichung bereitgestellt, Sie lernen jedoch, eine API-Gateway-REST-API zu konfigurieren, die jede Lambda-Ereignisfunktion aufrufen kann.



Die Verwendung von API Gateway bietet Benutzern einen sicheren HTTP-Endpunkt zum Aufrufen Ihrer Lambda-Ereignisfunktion und kann dazu beitragen, große Mengen an Aufrufen Ihrer Funktion zu verwalten, indem der Datenverkehr gedrosselt und API-Aufrufe automatisch validiert und autorisiert werden. API Gateway bietet auch flexible Sicherheitskontrollen mithilfe von AWS Identity and Access Management (IAM) und Amazon Cognito. Dies ist nützlich für Anwendungsfälle, in denen eine vorherige Autorisierung für Aufrufe zu Ihrer Anwendung erforderlich ist.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: API Gateway und Lambda-Funktions-URLs. Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “API Gateway im Vergleich zu Funktions-URLs”.

Um dieses Tutorial abzuschließen, durchlaufen Sie die folgenden Phasen:

1. Erstellen und konfigurieren Sie eine Lambda-Ereignisfunktion in Python oder Node.js um Operationen an einer DynamoDB-Tabelle durchzuführen.

2. Erstellen Sie eine REST-API in API Gateway, um eine Verbindung zu Ihrer Lambda-Ereignisfunktion herzustellen.
3. Erstellen Sie eine DynamoDB-Tabelle und testen Sie sie mit Ihrer Lambda-Ereignisfunktion in der Konsole.
4. Bereitstellen Ihrer API und Testen der vollständigen Einrichtung mithilfe von curl in einem Terminal.

Wenn Sie diese Phasen abschließen, lernen Sie, wie Sie mit API Gateway einen HTTP-Endpunkt erstellen, der eine Lambda-Ereignisfunktion in jeder Größenordnung sicher aufrufen kann. Sie lernen auch, wie Sie Ihre API bereitstellen und wie Sie sie in der Konsole und durch Senden einer HTTP-Anfrage über ein Terminal testen.

Erstellen einer Berechtigungsrichtlinie

Bevor Sie eine Ausführungsrolle für Ihre Lambda-Ereignisfunktion erstellen können, müssen Sie zunächst eine Berechtigungsrichtlinie erstellen, um Ihrer Funktion die Erlaubnis zu erteilen, auf die erforderlichen AWS Ressourcen zuzugreifen. In diesem Tutorial ermöglicht die Richtlinie Lambda, CRUD-Operationen in einer DynamoDB-Tabelle durchzuführen und in Amazon Logs zu schreiben. CloudWatch

So erstellen Sie die Richtlinie

1. Öffnen Sie die Seite Richtlinien in der IAM-Konsole.
2. Wählen Sie Richtlinie erstellen aus.
3. Wählen Sie die Registerkarte JSON aus und kopieren Sie dann die folgende benutzerdefinierte JSON-Richtlinie in den JSON-Editor.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
```

```
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. Wählen Sie Next: Tags (Weiter: Tags) aus.
5. Klicken Sie auf Weiter: Prüfen.
6. Geben Sie unter Review policy (Richtlinie prüfen) für den Richtlinien-Namen **`lambda-apigateway-policy`** ein.
7. Wählen Sie Richtlinie erstellen aus.

Erstellen einer Ausführungsrolle

Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Ereignisfunktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. Damit Ihre Funktion Vorgänge an einer DynamoDB-Tabelle ausführen kann, fügen Sie die Berechtigungsrichtlinie an, die Sie im vorherigen Schritt erstellt haben.

So erstellen Sie eine Ausführungsrolle und fügen Ihre benutzerdefinierte Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie als Typ der vertrauenswürdigen Entität AWS -Service und dann als Anwendungsfall Lambda aus.
4. Wählen Sie Weiter aus.

5. Geben Sie im Feld für die Richtliniensuche **`lambda-apigateway-policy`** ein.
6. Wählen Sie in den Suchergebnissen die von Ihnen erstellte Richtlinie (`lambda-apigateway-policy`) und dann die Option Next (Weiter) aus.
7. Geben Sie unter Role details (Rollendetails) für den Role name (Rollennamen) **`lambda-apigateway-role`** ein und wählen Sie dann Create role (Rolle erstellen) aus.

Erstellen Sie die Lambda-Ereignisfunktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Create Event Function.
2. Wählen Sie Von Grund auf neu schreiben aus.
3. Geben Sie für Function name (Funktionsname) `LambdaFunctionOverHttps` ein.
4. Wählen Sie für Runtime die neueste Node.js oder Python-Runtime aus.
5. Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern.
6. Wählen Sie Vorhandene Rolle verwenden und anschließend die zuvor erstellte Rolle „**`lambda-apigateway-role`**“ aus.
7. Wählen Sie „Ereignisfunktion erstellen“.
8. Ersetzen Sie im Bereich Codequelle den Standardcode durch den folgenden Code Node.js oder Python-Code.

   Node.js

   Die `region` Einstellung muss mit dem AWS-Region Ort übereinstimmen, an dem Sie die Funktion bereitstellen und die DynamoDB-Tabelle erstellen.

   Example index.mjs

```
import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-east-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
const tablename = "lambda-apigateway";
```

```
/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 *   - payload: a JSON object containing the parameters for the table item
 *     to perform the operation on
 */
export const handler = async (event, context) => {

     const operation = event.operation;

     if (operation == 'echo'){
          return(event.payload);
     }

    else {
        event.payload.TableName = tablename;
        let response;

        switch (operation) {
          case 'create':
               response = await ddbDocClient.send(new
 PutCommand(event.payload));
               break;
          case 'read':
               response = await ddbDocClient.send(new
 GetCommand(event.payload));
               break;
          case 'update':
               response = ddbDocClient.send(new UpdateCommand(event.payload));
               break;
          case 'delete':
               response = ddbDocClient.send(new DeleteCommand(event.payload));
               break;
          default:
            response = 'Unknown operation: ${operation}';
          }
        console.log(response);
        return response;
    }
};
```

Python

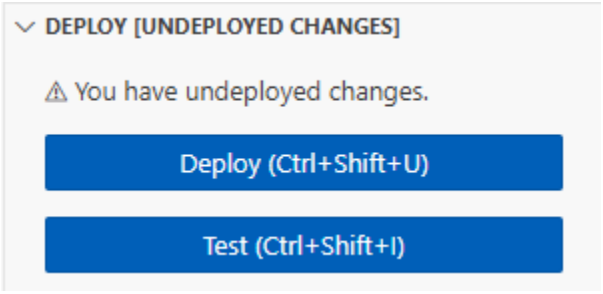
Example lambda_function.py

```
import boto3

# Define the DynamoDB table that Lambda will connect to
table_name = "lambda-apigateway"

# Create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(table_name)

# Define some functions to perform the CRUD operations
def create(payload):
    return dynamo.put_item(Item=payload['Item'])

def read(payload):
    return dynamo.get_item(Key=payload['Key'])

def update(payload):
    return dynamo.update_item(**{k: payload[k] for k in ['Key',
 'UpdateExpression',
    'ExpressionAttributeNames', 'ExpressionAttributeValues'] if k in payload})

def delete(payload):
    return dynamo.delete_item(Key=payload['Key'])

def echo(payload):
    return payload

operations = {
    'create': create,
    'read': read,
    'update': update,
    'delete': delete,
    'echo': echo,
}

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:
      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
        operation being performed
```

```
    '''

    operation = event['operation']
    payload = event['payload']

    if operation in operations:
        return operations[operation](payload)

    else:
        raise ValueError(f'Unrecognized operation "{operation}"')
```

> **Note**
>
> In diesem Beispiel ist der Name der DynamoDB-Tabelle als Variable in Ihrem Funktionscode definiert. In einer realen Anwendung besteht die bewährte Methode darin, diesen Parameter als Umgebungsvariable zu übergeben und den Tabellennamen nicht fest zu codieren. Weitere Informationen finden Sie unter AWS Lambda Umgebungsvariablen verwenden.

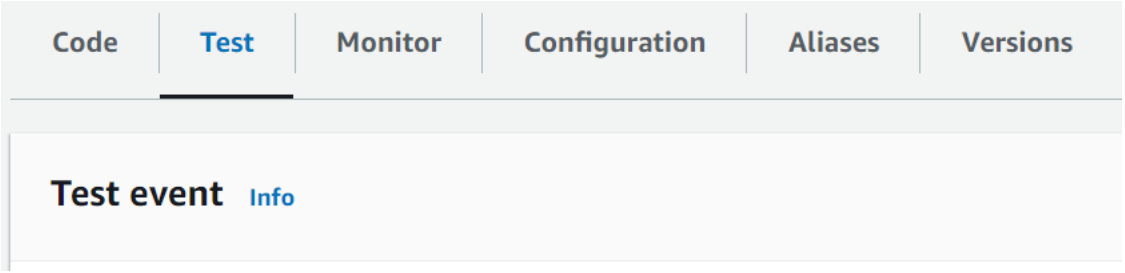
9. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Testen Sie die Event-Funktion

Bestätigen Sie vor der Integration Ihrer Funktion mit API-Gateway, dass Sie die Funktion erfolgreich bereitgestellt haben. Senden Sie mithilfe der Lambda-Konsole ein Testereignis an Ihre Funktion.

1. Wählen Sie auf der Lambda-Konsolenseite für Ihre Funktion die Registerkarte Test aus.



2. Scrollen Sie nach unten zum Abschnitt Ereignis-JSON und ersetzen Sie das Standardereignis durch Folgendes. Dieses Ereignis entspricht der Struktur, die von der Lambda-Ereignisfunktion erwartet wird.

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

3. Wählen Sie Test aus.
4. Erweitern Sie unter Funktionsausführung: erfolgreich die Option Details. Sie sollten die folgende Antwort sehen:

```
{
  "somekey1": "somevalue1",
  "somekey2": "somevalue2"
}
```

Erstellen Sie eine REST-API mit API Gateway

In diesem Schritt erstellen Sie die API-Gateway-REST-API, mit der Sie Ihre Lambda-Ereignisfunktion aufrufen.

So erstellen Sie die API

1. Öffnen Sie die API Gateway-Konsole.
2. Wählen Sie Create API (API erstellen) aus.
3. Wählen Sie im Feld REST-API die Option Entwickeln aus.

4. Lassen Sie unter API details (API-Details) die Option New API (Neue API) ausgewählt und geben Sie für API Name (API-Name) **`DynamoDBOperations`** ein.
5. Wählen Sie Create API (API erstellen) aus.

Erstellen einer Ressource für Ihre REST-API

Um Ihrer API eine HTTP-Methode hinzuzufügen, müssen Sie zunächst eine Ressource erstellen, mit der diese Methode ausgeführt werden kann. Hier erstellen Sie die Ressource zur Verwaltung Ihrer DynamoDB-Tabelle.

So erstellen Sie die Ressource

1. Wählen Sie in der API-Gateway-Konsole auf der Seite Ressourcen für Ihre API die Option Ressource erstellen aus.
2. Geben Sie im Feld Resource details (Ressourcendetails) für Resource name (Ressourcenname) **`DynamoDBManager`** ein.
3. Wählen Sie Create Resource (Ressource erstellen) aus.

Erstellen einer HTTP-POST-Methode

In diesem Schritt erstellen Sie eine Methode (`POST`) für Ihre `DynamoDBManager`-Ressource. Sie verknüpfen diese `POST` Methode mit Ihrer Lambda-Ereignisfunktion, sodass API Gateway Ihre Lambda-Ereignisfunktion aufruft, wenn die Methode eine HTTP-Anfrage empfängt.

> **Note**
>
> Für die Zwecke dieses Tutorials wird eine HTTP-Methode (`POST`) verwendet, um eine einzelne Lambda-Ereignisfunktion aufzurufen, die alle Operationen in Ihrer DynamoDB-Tabelle ausführt. In einer echten Anwendung empfiehlt es sich, für jeden Vorgang eine andere Lambda-Ereignisfunktion und HTTP-Methode zu verwenden. Weitere Informationen finden Sie bei Serverless Land unter The Lambda monolith.

So erstellen Sie die POST-Methode

1. Vergewissern Sie sich, dass die `/DynamoDBManager`-Ressource auf der Seite Resources (Ressourcen) für Ihre API markiert ist. Wählen Sie dann im Bereich Methods (Methoden) die Option Create Method (Methode erstellen) aus.

2. Wählen Sie in Method type (Methodentyp) POST.
3. Lassen Sie für den Integration type (Integrationstyp) die Option Lambda function (Lambda-Funktion) ausgewählt.
4. Wählen Sie für die Lambda function (Lambda-Funktion) den Amazon Ressourcennamen (ARN) für Ihre Funktion (`LambdaFunctionOverHttps`).
5. Wählen Sie Methode erstellen aus.

Erstellen einer DynamoDB-Tabelle

Erstellen Sie eine leere DynamoDB-Tabelle, für die Ihre Lambda-Ereignisfunktion CRUD-Operationen ausführt.

Erstellen einer DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie Create table (Tabelle erstellen) aus.
3. Führen Sie unter Tabellendetails die folgenden Schritte aus:
    1. Geben Sie unter Tabellenname **`lambda-apigateway`** ein.
    2. Geben Sie für Partitionsschlüssel **`id`** ein und behalten Sie den Datentyp als Zeichenfolge bei.
4. Behalten Sie unter Table settings (Tabelleneinstellungen) die Default settings (Standardeinstellungen) bei.
5. Wählen Sie Create table (Tabelle erstellen) aus.

Testen der Integration von API-Gateway, Lambda und DynamoDB

Sie sind jetzt bereit, die Integration Ihrer API-Gateway-API-Methode mit Ihrer Lambda-Ereignisfunktion und Ihrer DynamoDB-Tabelle zu testen. Mit der API Gateway-Konsole senden Sie Anfragen mithilfe der Testfunktion der Konsole direkt an Ihre `POST`-Methode. In diesem Schritt verwenden Sie zuerst eine `create`-Operation, um Ihrer DynamoDB-Tabelle ein neues Element hinzuzufügen, und verwenden dann eine `update`-Operation, um das Element zu ändern.

Test 1: So erstellen Sie ein neues Element in Ihrer DynamoDB-Tabelle

1. Wählen Sie in der API-Gateway-Konsole Ihre API aus (`DynamoDBOperations`).
2. Wählen Sie unter der Ressource `DynamoDBManager` die Methode POST.

3. Wählen Sie die Registerkarte Test. Möglicherweise müssen Sie die rechte Pfeiltaste wählen, um die Registerkarte anzuzeigen.
4. Lassen Sie unter Test method (Testmethode) die Felder Query strings (Query-Strings) und Headers (Header) leer. Fügen Sie für Request body (Anforderungstext) den folgenden JSON-Code ein:

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. Wählen Sie Test aus.

   Die Ergebnisse, die nach Abschluss des Tests angezeigt werden, sollten den Status `200` anzeigen. Dieser Statuscode zeigt an, dass die `create`-Operation erfolgreich war.

   Überprüfen Sie zur Bestätigung, ob Ihre DynamoDB-Tabelle jetzt das neue Element enthält.
6. Öffnen Sie die Seite Tabellen der DynamoDB-Konsole und wählen Sie die `lambda-apigateway`-Tabelle aus.
7. Wählen Sie Explore table items (Tabellenelemente erkunden) aus. Im Bereich Items returned (Zurückgegebene Elemente) sollten ein Element mit der id (ID) `1234ABCD` und der number (Nummer) `5` angezeigt werden. Beispiel:



Test 2: So aktualisieren Sie das Element in Ihrer DynamoDB-Tabelle

1. Kehren Sie in der API-Gateway-Konsole zum Bereich Test Ihrer POST-Methode zurück.

2. Lassen Sie unter Test method (Testmethode) die Felder Query strings (Query-Strings) und Headers (Header) leer. Fügen Sie für Request body (Anforderungstext) den folgenden JSON-Code ein:

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "UpdateExpression": "SET #num = :newNum",
        "ExpressionAttributeNames": {
            "#num": "number"
        },
        "ExpressionAttributeValues": {
            ":newNum": 10
        }
    }
}
```

3. Wählen Sie Test aus.

   Die Ergebnisse, die nach Abschluss des Tests angezeigt werden, sollten den Status `200` anzeigen. Dieser Statuscode zeigt an, dass die `update`-Operation erfolgreich war.

   Überprüfen Sie zur Bestätigung, ob das Element in Ihrer DynamoDB-Tabelle geändert wurde.

4. Öffnen Sie die Seite Tabellen der DynamoDB-Konsole und wählen Sie die `lambda-apigateway`-Tabelle aus.

5. Wählen Sie Explore table items (Tabellenelemente erkunden) aus. Im Bereich Items returned (Zurückgegebene Elemente) sollten ein Element mit der id (ID) `1234ABCD` und der number (Nummer) `10` angezeigt werden.



Bereitstellen der API

Damit ein Client die API aufrufen kann, müssen Sie eine Bereitstellung und eine zugehörige Stufe erstellen. Eine Phase stellt eine Momentaufnahme Ihrer API dar, einschließlich ihrer Methoden und Integrationen.

So stellen Sie die API bereit

1. Öffnen Sie die APIs-Seite der API-Gateway-Konsole und wählen Sie die `DynamoDBOperations`-API aus.
2. Wählen Sie auf der Seite Resources (Ressourcen) für Ihre API die Option Deploy API (API bereitstellen) aus.
3. Wählen Sie für Stage (Stufe) *New stage* (*Neue Stufe*) und geben Sie dann als Stage name (Phasenname) **`test`** ein.
4. Wählen Sie Bereitstellen.
5. Kopieren Sie im Bereich Stage details (Stufendetails) die Invoke URL (Aufruf-URL). Sie verwenden dies im nächsten Schritt, um Ihre Funktion mithilfe einer HTTP-Anfrage aufzurufen.

Verwenden Sie curl, um Ihre Event-Funktion mithilfe von HTTP-Anfragen aufzurufen

Sie können jetzt Ihre Lambda-Ereignisfunktion aufrufen, indem Sie eine HTTP-Anfrage an Ihre API senden. In diesem Schritt erstellen Sie ein neues Element in Ihrer DynamoDB-Tabelle und führen dann Lese-, Aktualisierungs- und Löschvorgänge für dieses Element durch.

So erstellen Sie mit curl ein Element in Ihrer DynamoDB-Tabelle

1. Öffnen Sie ein Terminal oder eine Eingabeaufforderung auf Ihrem lokalen Computer und führen Sie den folgenden `curl`-Befehl mit der im vorherigen Schritt kopierten Aufruf-URL aus. Dieser Befehl verwendet die folgenden Optionen:

   - `-H`: Fügt der Anforderung einen benutzerdefinierten Header hinzu. Hier wird der Inhaltstyp als JSON angegeben.
   - `-d`: Sendet Daten im Anforderungstext. Diese Option verwendet standardmäßig eine HTTP-POST-Methode.

Linux/macOS

```
curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
 \
-H "Content-Type: application/json" \
-d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number":
 15}}}'
```

PowerShell

```
curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
 \"create\", \"payload\": {\"Item\": {\"id\": \"5678EFGH\", \"number\": 15}}}'
```

Wenn der Vorgang erfolgreich war, sollten Sie eine Antwort mit einem HTTP-Statuscode von 200 erhalten.

2. Sie können auch die DynamoDB-Konsole verwenden, um zu überprüfen, ob sich das neue Element in Ihrer Tabelle befindet, indem Sie Folgendes tun:

   1. Öffnen Sie die Seite Tabellen in der DynamoDB-Konsole und wählen Sie die `lambda-apigateway`-Tabelle aus.
   2. Wählen Sie Explore Table Items (Tabellenelemente erkudnen) aus. Im Bereich Items returned (Zurückgegebene Elemente) sollte ein Element mit der id (ID) `5678EFGH` und der number (Nummer) 15 angezeigt werden.

So lesen Sie das Element in Ihrer DynamoDB-Tabelle mit curl

- Führen Sie in Ihrem Terminal oder Ihrer Eingabeaufforderung den folgenden `curl`-Befehl aus, um den Wert des soeben erstellten Elements zu lesen. Verwenden Sie Ihre eigene Aufruf-URL.

  Linux/macOS

```
curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
 \
-H "Content-Type: application/json" \
-d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
```

PowerShell

```
curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
```

Sie sollten eine Ausgabe wie eine der folgenden sehen, je nachdem, ob Sie den Funktionscode Node.js oder den Python-Funktionscode ausgewählt haben:

Node.js

```
{"$metadata":
{"httpStatusCode":200,"requestId":"7BP3G5Q0C0O1E50FBQI9NS099JVV4KQNSO5AEMVJF66Q9ASUAAJG"
"attempts":1,"totalRetryDelay":0},"Item":{"id":"5678EFGH","number":15}}
```

Python

```
{"Item":{"id":"5678EFGH","number":15},"ResponseMetadata":
{"RequestId":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG",
"HTTPStatusCode":200,"HTTPHeaders":{"server":"Server","date":"Wed, 31 Jul 2024
 00:37:01 GMT","content-type":"application/x-amz-json-1.0",
"content-length":"52","connection":"keep-alive","x-amzn-
requestid":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG","x-amz-
crc32":"2589610852"},
"RetryAttempts":0}}
```

So aktualisieren Sie das Element in Ihrer DynamoDB-Tabelle mit curl

1. Führen Sie in Ihrem Terminal oder Ihrer Eingabeaufforderung den folgenden `curl`-Befehl aus, um das soeben erstellte Element durch Ändern des `number`-Werts zu aktualisieren. Verwenden Sie Ihre eigene Aufruf-URL.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   ```

```
-d '{"operation": "update", "payload": {"Key": {"id": "5678EFGH"},
 "UpdateExpression": "SET #num = :new_value", "ExpressionAttributeNames":
 {"#num": "number"}, "ExpressionAttributeValues": {":new_value": 42}}}'
```

PowerShell

```
curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
 \"update\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}, \"UpdateExpression
\": \"SET #num = :new_value\", \"ExpressionAttributeNames\": {\"#num\": \"number
\"}, \"ExpressionAttributeValues\": {\":new_value\": 42}}}'
```

2. Um zu bestätigen, dass der Wert von `number` für das Element aktualisiert wurde, führen Sie einen weiteren Lesebefehl aus:

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
   \", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
   ```

So löschen Sie das Element in Ihrer DynamoDB-Tabelle mit curl

1. Führen Sie in Ihrem Terminal oder Ihrer Eingabeaufforderung den folgenden `curl`-Befehl aus, um das soeben erstellte Element zu löschen. Verwenden Sie Ihre eigene Aufruf-URL.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

PowerShell

```
curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
 \"delete\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
```

2. Bestätigen Sie, dass der Löschvorgang erfolgreich war. Stellen Sie im Bereich Zurückgegebene Elemente der Seite Elemente entdecken der DynamoDB-Konsole sicher, dass das Element mit der ID `5678EFGH` nicht mehr in der Tabelle enthalten ist.

Bereinigen Ihrer Ressourcen (optional)

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Indem Sie AWS Ressourcen löschen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die API

1. Öffnen Sie die API-Seite der API-Gateway-Konsole.
2. Wählen Sie die von Ihnen erstellte API aus.
3. Wählen Sie Aktionen, Löschen aus.

4. Wählen Sie Löschen aus.

So löschen Sie die DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie die von Ihnen erstellte Tabelle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie **`delete`** in das Textfeld ein.
5. Wählen Sie Delete Table (Tabelle löschen).

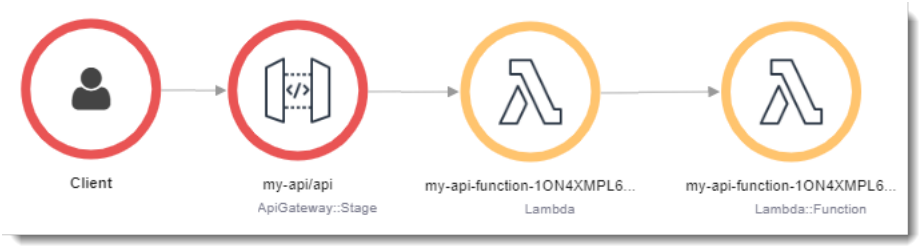
Wenn beim Aufrufen Ihrer Funktion über API Gateway Fehler auftreten, finden Sie weitere Informationen unterthe section called “Fehler”.

Behandlung von Lambda-Fehlern mit einer API-Gateway-API

API Gateway behandelt alle Aufruf- und Eventfunktionsfehler als interne Fehler. Wenn die Lambda-API die Aufruf-Anforderung ablehnt, gibt API Gateway einen 500-Fehlercode zurück. Wenn die Event-Funktion ausgeführt wird, aber einen Fehler zurückgibt oder eine Antwort im falschen Format zurückgibt, gibt API Gateway 502 zurück. In beiden Fällen lautet der Text der Antwort von API Gateway „`{"message": "Internal server error"}`“.

> **Note**
>
> API Gateway wiederholt keine Lambda-Aufrufe. Wenn Lambda einen Fehler zurückgibt, gibt API Gateway eine Fehlerantwort an den Client zurück.

Das folgende Beispiel zeigt eine X-Ray Trace-Map für eine Anfrage, die zu einem Funktionsfehler geführt hat, und einen 502 von API Gateway. Der Client erhält die generische Fehlermeldung.

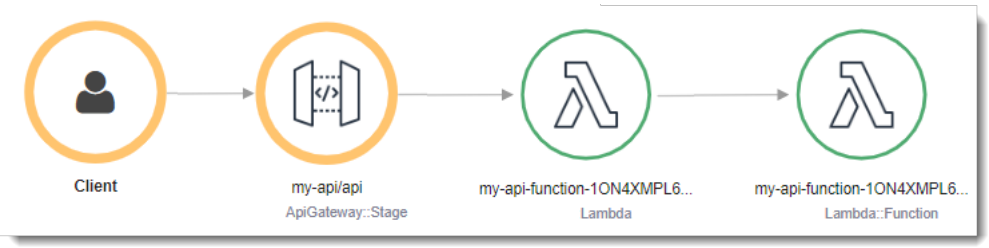


Um die Fehlerantwort anzupassen, müssen Sie Fehler im Code abfangen und eine Antwort im erforderlichen Format formatieren.

Example index.mjs — Fehler beim Formatieren

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway konvertiert diese Antwort in einen HTTP-Fehler mit einem benutzerdefinierten Statuscode und Text. In der Ablaufverfolgungszuweisung ist der Funktionsknoten grün, da er den Fehler behandelt hat.



Wählen Sie eine Methode aus, um Ihre Lambda-Ereignisfunktion mithilfe einer HTTP-Anfrage aufzurufen

In vielen gängigen Anwendungsfällen für Lambda wird Ihre Event-Funktion mithilfe einer HTTP-Anfrage aufgerufen. Beispielsweise möchten Sie möglicherweise, dass eine Webanwendung Ihre Funktion über eine Browseranforderung aufruft. Lambda Event Functions können auch verwendet werden, um vollständige REST-APIs zu erstellen, Benutzerinteraktionen aus mobilen Apps zu verarbeiten, Daten von externen Diensten über HTTP-Aufrufe zu verarbeiten oder benutzerdefinierte Webhooks zu erstellen.

In den folgenden Abschnitten wird erläutert, welche Möglichkeiten Sie haben, Lambda über HTTP aufzurufen und es werden Informationen bereitgestellt, die Ihnen helfen, die richtige Entscheidung für Ihren speziellen Anwendungsfall zu treffen.

Welche Optionen stehen Ihnen bei der Auswahl einer HTTP-Aufrufmethode zur Verfügung?

Lambda bietet zwei Hauptmethoden zum Aufrufen einer Funktion mithilfe einer HTTP-Anfrage: Funktions-URLs und API-Gateway. Die wichtigsten Unterschiede zwischen diesen beiden Optionen sind die folgenden:

- Lambda-Funktions-URLs bieten einen einfachen, direkten HTTP-Endpunkt für eine Lambda-Ereignisfunktion. Sie sind auf Einfachheit und Wirtschaftlichkeit optimiert und bieten den schnellsten Weg, eine Lambda-Ereignisfunktion über HTTP verfügbar zu machen.
- API Gateway ist ein fortschrittlicherer Dienst zum Erstellen von APIs mit vollem Funktionsumfang. API Gateway ist für die Erstellung und Verwaltung von Produktions-APIs in großem Maßstab optimiert und bietet umfassende Tools für Sicherheit, Überwachung und Datenverkehrsmanagement.

Empfehlungen, falls Sie Ihre Anforderungen bereits kennen

Wenn Sie sich über Ihre Anforderungen bereits im Klaren sind, finden Sie hier unsere grundlegenden Empfehlungen:

Wir empfehlen Funktions-URLs für einfache Anwendungen oder das Prototyping, bei denen Sie nur grundlegende Authentifizierungsmethoden und deren request/response Handhabung benötigen und bei denen Sie Kosten und Komplexität auf ein Minimum reduzieren möchten.

API Gateway ist eine bessere Wahl für Produktionsanwendungen in großem Maßstab oder für Fälle, in denen Sie erweiterte Funktionen wie OpenAPI-Description, eine Auswahl an Authentifizierungsoptionen, benutzerdefinierte Domainnamen oder umfangreiche request/response Funktionen wie Drosselung, Caching und Transformation benötigen. request/response

Was Sie bei der Auswahl einer Methode zum Aufrufen Ihrer Lambda-Ereignisfunktion beachten sollten

Bei der Wahl zwischen Funktions-URLs und API-Gateway müssen Sie die folgenden Faktoren berücksichtigen:

- Ihre Authentifizierungsanforderungen, z. B. ob Sie OAuth oder Amazon Cognito zur Benutzerauthentifizierung benötigen

- Ihre Skalierungsanforderungen und die Komplexität der API, die Sie implementieren möchten
- Ob Sie erweiterte Funktionen wie die Validierung und Formatierung von Anfragen benötigen request/response
- Ihre Überwachungsanforderungen
- Was sind Ihre Ziele?

Wenn Sie diese Faktoren verstehen, können Sie die Option auswählen, die Ihren Anforderungen an Sicherheit, Komplexität und Kosten am besten gerecht wird.

Die folgenden Informationen fassen die wichtigsten Unterschiede zwischen den beiden Optionen zusammen.

Authentifizierung

- Funktions-URLs bieten grundlegende Authentifizierungsoptionen über AWS Identity and Access Management (IAM). Sie können Ihre Endpunkte so konfigurieren, dass sie entweder öffentlich sind (keine Authentifizierung) oder dass sie eine IAM-Authentifizierung erfordern. Mit der IAM-Authentifizierung können Sie AWS Standardanmeldedaten oder IAM-Rollen verwenden, um den Zugriff zu steuern. Dieser Ansatz ist zwar einfach einzurichten, bietet aber im Vergleich zu anderen Authentifizierungsmethoden nur begrenzte Optionen.
- API Gateway bietet Zugriff auf eine umfassendere Palette von Authentifizierungsoptionen. Neben der IAM-Authentifizierung können Sie Lambda-Autorisierer (benutzerdefinierte Authentifizierungslogik), Amazon Cognito Cognito-Benutzerpools und Flows verwenden. OAuth2.0 Diese Flexibilität ermöglicht es Ihnen, komplexe Authentifizierungsschemata zu implementieren, darunter Authentifizierungsanbieter von Drittanbietern, tokenbasierte Authentifizierung und Multi-Faktor-Authentifizierung.

Request/response Handhabung

- Funktions-URLs ermöglichen die grundlegende Behandlung von HTTP-Anfragen und -Antworten. Sie unterstützen Standard-HTTP-Methoden und bieten integrierte CORS-Unterstützung (Cross-Origin Resource Sharing). Sie können zwar JSON-Nutzdaten und Abfrageparameter auf natürliche Weise verarbeiten, bieten jedoch keine Funktionen zur Transformation oder Validierung von Anfragen. Die Bearbeitung von Antworten ist ähnlich einfach — der Client erhält die Antwort von Ihrer Lambda-Ereignisfunktion genau so, wie Lambda sie zurückgibt.

- API Gateway bietet ausgefeilte Funktionen zur Bearbeitung von Anfragen und Antworten. Sie können Anforderungsvalidatoren definieren, Anfragen und Antworten mithilfe von Zuordnungsvorlagen transformieren, request/response Header einrichten und das Zwischenspeichern von Antworten implementieren. API Gateway unterstützt auch binäre Nutzdaten und benutzerdefinierte Domainnamen und kann Antworten ändern, bevor sie den Client erreichen. Sie können Modelle für die request/response Validierung und Transformation mithilfe des JSON-Schemas einrichten.

Skalierung

- Funktions-URLs passen sich direkt an die Parallelitätsgrenzen Ihrer Lambda-Ereignisfunktion an und bewältigen Verkehrsspitzen, indem Sie Ihre Funktion bis zum konfigurierten Maximallimit skalieren. Sobald dieses Limit erreicht ist, reagiert Lambda auf weitere Anfragen mit HTTP 429-Antworten. Es gibt keinen integrierten Warteschlangenmechanismus, sodass der Umgang mit der Skalierung vollständig von der Konfiguration Ihrer Lambda-Ereignisfunktion abhängt. Standardmäßig haben Lambda-Ereignisfunktionen ein Limit von 1.000 gleichzeitigen Ausführungen pro. AWS-Region
- API Gateway bietet zusätzlich zur Lambda-eigenen Skalierung zusätzliche Skalierungsfunktionen. Es umfasst integrierte Funktionen zur Warteschleife und Drosselung von Anfragen, sodass Sie Datenverkehrsspitzen besser bewältigen können. API Gateway kann standardmäßig bis zu 10.000 Anfragen pro Sekunde pro Region verarbeiten, mit einer Burst-Kapazität von 5.000 Anfragen pro Sekunde. Es bietet auch Tools zum Drosseln von Anfragen auf verschiedenen Ebenen (API, Phase oder Methode), um Ihr Backend zu schützen.

Überwachen

- Funktions-URLs ermöglichen eine grundlegende Überwachung anhand von CloudWatch Amazon-Metriken, einschließlich Anzahl der Anfragen, Latenz und Fehlerraten. Sie erhalten Zugriff auf standardmäßige Lambda-Metriken und -Protokolle, die die Rohanfragen anzeigen, die in Ihre Funktion eingehen. Dies bietet zwar wichtige betriebliche Transparenz, die Metriken konzentrieren sich jedoch hauptsächlich auf die Funktionsausführung.
- API Gateway bietet umfassende Überwachungsfunktionen, einschließlich detaillierter Metriken, Protokollierungs- und Verfolgungsoptionen. Sie können API-Aufrufe, Latenz, Fehlerraten und hit/miss Cache-Raten über überwachen CloudWatch. API Gateway lässt sich auch AWS X-Ray für verteiltes Tracing integrieren und bietet anpassbare Protokollierungsformate.

Cost (Kosten)

- Funktions-URLs folgen dem Standard-Lambda-Preismodell – Sie zahlen nur für Funktionsaufrufe und Rechenzeit. Für den URL-Endpunkt selbst fallen keine zusätzlichen Gebühren an. Dies macht es zu einer kostengünstigen Wahl für einfache APIs oder Anwendungen mit geringem Datenverkehr, wenn Sie die zusätzlichen Features von API Gateway nicht benötigen.
- API Gateway bietet ein kostenloses Kontingent, das eine Million API-Aufrufe für REST-APIs und eine Million API-Aufrufe für HTTP-APIs umfasst. Danach berechnet API Gateway Gebühren für API-Aufrufe, Datenübertragung und Caching (falls aktiviert). Weitere Informationen zu den Kosten für Ihren eigenen Anwendungsfall finden Sie auf der Seite mit den Preisen für API Gateway.

Weitere Funktionen

- Funktions-URLs sind auf Einfachheit und direkte Lambda-Integration ausgelegt. Sie unterstützen sowohl HTTP- als auch HTTPS-Endpunkte, bieten integrierte CORS-Unterstützung und bieten Dual-Stack-Endpunkte (IPv4 und IPv6). Ihnen fehlen zwar erweiterte Funktionen, sie zeichnen sich jedoch in Szenarien aus, in denen Sie Lambda-Ereignisfunktionen schnell und unkompliziert über HTTP verfügbar machen müssen.
- API Gateway umfasst zahlreiche zusätzliche Funktionen wie API-Versionierung, Stufenmanagement, API-Schlüssel für Nutzungspläne, API-Dokumentation über Swagger/OpenAPI, WebSocket APIs, private APIs innerhalb einer VPC und WAF-Integration für zusätzliche Sicherheit. Es unterstützt auch kanarische Implementierungen, Scheinintegrationen zum Testen und die Integration mit anderen Systemen AWS-Services als Lambda.

Wählen Sie eine Methode zum Aufrufen Ihrer Lambda-Ereignisfunktion

Nachdem Sie sich mit den Kriterien für die Auswahl zwischen Lambda-Funktions-URLs und API Gateway und den wichtigsten Unterschieden zwischen ihnen vertraut gemacht haben, können Sie die Option auswählen, die Ihren Anforderungen am besten entspricht und die folgenden Ressourcen verwenden, um Ihnen den Einstieg zu erleichtern.

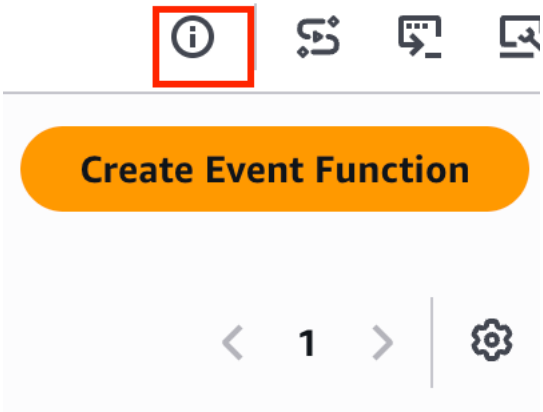
Function URLs

Nutzen Sie die folgenden Ressourcen, um mit Funktions-URLs zu beginnen

- Folgen Sie dem Tutorial Erstellen einer Lambda-Ereignisfunktion mit einer Funktions-URL

- Weitere Informationen zu Funktions-URLs finden Sie im the section called “Funktions-URLs”-Kapitel dieses Handbuchs
- Probieren Sie das in der Konsole enthaltene Tutorial Erstellen Sie eine einfache Web-App aus, indem Sie wie folgt vorgehen:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Öffnen Sie das Hilfe-Panel, indem Sie das Symbol in der oberen rechten Ecke des Bildschirms wählen.



3. Wählen Sie Tutorials aus.
4. Wählen Sie unter Einfache Web-App erstellen die Option Tutorial starten aus.

API Gateway

Beginnen Sie mit Lambda und API Gateway mit den folgenden Ressourcen

- Folgen Sie dem Tutorial Verwenden von Lambda mit API Gateway, um eine REST-API zu erstellen, die in eine Lambda-Ereignisfunktion im Backend integriert ist.
- In den folgenden Abschnitten des Amazon API Gateway Developer Guide erfahren Sie mehr über die verschiedenen Arten von APIs, die von API Gateway angeboten werden:
  - API Gateway-REST-APIs
  - API Gateway-HTTP-APIs
  - WebSocket API-Gateway-APIs
- Probieren Sie eines oder mehrere der Beispiele im Abschnitt Tutorials und Workshops im Amazon API Gateway Developer Guide aus.

Verwenden AWS Lambda mit AWS Infrastructure Composer

AWS Infrastructure Composer ist ein Visual Builder zum Entwerfen moderner Anwendungen auf AWS. Sie entwerfen Ihre Anwendungsarchitektur, indem Sie sie auf einer visuellen Leinwand ziehen, gruppieren und verbinden AWS-Services . Infrastructure Composer erstellt anhand Ihres Entwurfs IaC-Vorlagen (Infrastructure as Code), die Sie mithilfe von AWS SAM oder CloudFormation bereitstellen können.

Exportieren einer Lambda-Ereignisfunktion nach Infrastructure Composer

Sie können mit Infrastructure Composer beginnen, indem Sie mithilfe der Lambda-Konsole ein neues Projekt erstellen, das auf der Konfiguration einer vorhandenen Lambda-Ereignisfunktion basiert. Gehen Sie wie folgt vor, um die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren und ein neues Projekt zu erstellen:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie als Grundlage für Ihr Infrastructure-Composer-Projekt verwenden möchten.
3. Wählen Sie im Bereich Funktionsübersicht die Option Nach Infrastructure Composer exportieren aus.

   Um die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren, erstellt Lambda einen Amazon-S3-Bucket in Ihrem Konto, um diese Daten vorübergehend zu speichern.
4. Wählen Sie im Dialogfeld Projekt bestätigen und erstellen aus, um den Standardnamen für diesen Bucket zu akzeptieren und die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren.
5. (Optional) Um einen anderen Namen für den von Lambda erstellten Amazon-S3-Bucket auszuwählen, geben Sie einen neuen Namen ein und wählen Sie Projekt bestätigen und erstellen aus. Die Amazon-S3-Bucket-Namen müssen global eindeutig sein und den Regeln für die Benennung von Buckets entsprechen.
6. Um Ihre Projekt- und Funktionsdateien in Infrastructure Composer zu speichern, aktivieren Sie den lokalen Synchronisierungsmodus.

> **Note**
>
> Wenn Sie das Feature Nach Application Composer exportieren bereits verwendet und einen Amazon-S3-Bucket mit dem Standardnamen erstellt haben, kann Lambda diesen Bucket wiederverwenden, falls er noch existiert. Akzeptieren Sie den Standard-Bucket-Namen im Dialogfeld, um den vorhandenen Bucket wiederzuverwenden.

Konfiguration des Amazon-S3-Transfer-Buckets

Der Amazon-S3-Bucket, den Lambda für die Übertragung der Konfiguration Ihrer Funktion erstellt, verschlüsselt Objekte automatisch mit dem Verschlüsselungsstandard AES 256. Lambda konfiguriert den Bucket auch so, dass er die Bucket-Besitzer-Bedingung verwendet, um sicherzustellen, dass nur Sie AWS-Konto Objekte zum Bucket hinzufügen können.

Lambda konfiguriert den Bucket so, dass Objekte 10 Tage nach dem Hochladen automatisch gelöscht werden. Lambda löscht den Bucket selbst jedoch nicht automatisch. Um den Bucket aus Ihrem zu löschen AWS-Konto, folgen Sie den Anweisungen unter Bucket löschen. Der Standard-Bucket-Name verwendet das Präfix`lambdasam`, eine 10-stellige alphanumerische Zeichenfolge und das, in dem AWS-Region Sie Ihre Funktion erstellt haben:

```
lambdasam-06f22da95b-us-east-1
```

Um zusätzliche Gebühren zu vermeiden, empfehlen wir Ihnen AWS-Konto, den Amazon S3 S3-Bucket zu löschen, sobald Sie den Export Ihrer Funktion in Infrastructure Composer abgeschlossen haben.

Es gelten die Standardpreise von Amazon S3.

Erforderliche Berechtigungen

Um die Funktion Lambda-Integration mit Infrastructure Composer verwenden zu können, benötigen Sie bestimmte Berechtigungen, um eine AWS SAM Vorlage herunterzuladen und die Konfiguration Ihrer Funktion in Amazon S3 zu schreiben.

Um eine AWS SAM Vorlage herunterzuladen, benötigen Sie die Erlaubnis, die folgenden API-Aktionen zu verwenden:

- GetPolicy

- Ich bin: GetPolicyVersion
- ich bin: GetRole
- ich bin: GetRolePolicy
- ich bin: ListAttachedRolePolicies
- ich bin: ListRolePolicies
- ich bin: ListRoles

Sie können die Erlaubnis zur Verwendung all dieser Aktionen erteilen, indem Sie die von AWS Lambda_ReadOnlyAccess AWS verwaltete Richtlinie zu Ihrer IAM-Benutzerrolle hinzufügen.

Damit Lambda die Konfiguration Ihrer Funktion in Amazon S3 schreiben kann, benötigen Sie die Erlaubnis, die folgenden API-Aktionen zu verwenden:

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption
- S3:PutBucketLifecycleConfiguration

Wenn Sie die Konfiguration Ihrer Funktion nicht nach Infrastructure Composer exportieren können, überprüfen Sie, ob Ihr Konto über die erforderlichen Berechtigungen für diese Operationen verfügt. Wenn Sie über die erforderlichen Berechtigungen verfügen, die Konfiguration Ihrer Funktion aber immer noch nicht exportieren können, suchen Sie nach ressourcenbasierten Richtlinien, die möglicherweise den Zugriff auf Amazon S3 einschränken.

Sonstige Ressourcen

Ein ausführlicheres Tutorial zum Entwerfen einer serverlosen Anwendung in Infrastructure Composer auf der Grundlage einer vorhandenen Lambda-Ereignisfunktion finden Sie unter. the section called “Infrastructure as Code (IaC)”

Um Infrastructure Composer AWS SAM zu verwenden und eine vollständige serverlose Anwendung mit Lambda zu entwerfen und bereitzustellen, können Sie auch dem AWS Infrastructure Composer Tutorial im AWS Serverless Patterns Workshop folgen.

Verwenden AWS Lambda mit CloudFormation

In einer AWS CloudFormation Vorlage können Sie eine Lambda-Ereignisfunktion als Ziel einer benutzerdefinierten Ressource angeben. Verwenden Sie benutzerdefinierte Ressourcen, um Parameter zu verarbeiten, Konfigurationswerte abzurufen oder andere AWS-Services während Stack-Lifecycle-Ereignissen aufzurufen.

Das folgende Beispiel ruft eine Funktion auf, die an anderer Stelle in der Vorlage definiert ist.

Example– benutzerdefinierte Ressourcendefinition

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

Das Service-Token ist der Amazon-Ressourcenname (ARN) der Funktion, die CloudFormation aufgerufen wird, wenn Sie den Stack erstellen, aktualisieren oder löschen. Sie können auch zusätzliche Eigenschaften hinzufügen`FunctionName`, wie z. B., was CloudFormation unverändert an Ihre Funktion übergeben wird.

CloudFormation ruft Ihre Lambda-Event-Funktion asynchron mit einem Ereignis auf, das eine Callback-URL enthält.

Example– CloudFormation Nachrichtenereignis

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
```

```
    "LogicalResourceId": "primerinvoke",
    "ResourceType": "AWS::CloudFormation::CustomResource",
    "ResourceProperties": {
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

Die Funktion ist für die Rückgabe einer Antwort an die Rückruf-URL zuständig, die Erfolg oder Misserfolg bedeutet. Die vollständige Antwortsyntax finden Sie unter Benutzerdefinierte Ressourcenantwortobjekte.

Example– CloudFormation benutzerdefinierte Ressourcenantwort

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

CloudFormation stellt eine Bibliothek namens bereit`cfn-response`, die das Senden der Antwort übernimmt. Wenn Sie Ihre Funktion innerhalb einer Vorlage definieren, können Sie die Bibliothek anhand des Namens angeben. CloudFormation fügt dann die Bibliothek dem Bereitstellungspaket hinzu, das sie für die Funktion erstellt.

Wenn Ihre Funktion, die eine benutzerdefinierte Ressource verwendet, mit einer Elastic Network Interface verbunden ist, fügen Sie die folgenden Ressourcen zur VPC-Richtlinie hinzu, wobei **`region`** die Region ist, in der sich die Funktion befindet, ohne Bindestriche angegeben ist. Zum Beispie, `us-east-1` ist `useast1`. Dadurch kann die benutzerdefinierte Ressource auf die Callback-URL antworten, die ein Signal zurück an den CloudFormation Stack sendet.

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

Die folgende Beispielfunktion ruft eine zweite Funktion auf. Wenn der Aufruf erfolgreich ist, sendet die Funktion eine Erfolgsantwort an CloudFormation, und das Stack-Update wird fortgesetzt. Die

Vorlage verwendet den Ressourcentyp AWS::Serverless: :Function, der von bereitgestellt wird. AWS Serverless Application Model

Example– Benutzerdefinierte Ressourcenfunktion

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

Wenn die Funktion, die die benutzerdefinierte Ressource aufruft, nicht in einer Vorlage definiert ist, können Sie den Quellcode für `cfn-response` das Modul cfn-response im Benutzerhandbuch abrufen. AWS CloudFormation

Weitere Informationen zu benutzerdefinierten Ressourcen finden Sie unter Benutzerdefinierte Ressourcen im AWS CloudFormation -Benutzerhandbuch.

Amazon DocumentDB-Ereignisse mit Lambda verarbeiten

Sie können eine Lambda-Ereignisfunktion verwenden, um Ereignisse in einem Amazon DocumentDB-Change-Stream (mit MongoDB-Kompatibilität) zu verarbeiten, indem Sie einen Amazon DocumentDB-Cluster als Ereignisquelle konfigurieren. Anschließend können Sie ereignisgesteuerte Workloads automatisieren, indem Sie Ihre Lambda Event Function jedes Mal aufrufen, wenn sich die Daten in Ihrem Amazon DocumentDB-Cluster ändern.

> **Note**
>
> Lambda unterstützt nur die Versionen 4.0 und 5.0 von Amazon DocumentDB. Lambda unterstützt Version 3.6 nicht.
> Für die Zuordnung von Ereignisquellen unterstützt Lambda außerdem nur Instance-basierte Cluster und regionale Cluster. Lambda unterstützt keine elastischen Cluster oder globalen Cluster. Diese Einschränkung gilt nicht, wenn Lambda als Client verwendet wird, um eine Verbindung mit Amazon DocumentDB herzustellen. Lambda kann eine Verbindung mit allen Clustertypen herstellen, um CRUD-Vorgänge auszuführen.

Lambda verarbeitet Ereignisse aus Amazon-DocumentDB-Änderungsstreams sequentiell in der Reihenfolge, in der sie ankommen. Aus diesem Grund kann Ihre Funktion jeweils nur einen Aufruf von Amazon DocumentDB gleichzeitig verarbeiten. Sie können Ihre Funktion anhand ihrer Gleichzeitigkeitsmetriken überwachen.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Topics

- Beispiel für Amazon-DocumentDB-Ereignis
- Voraussetzungen und Berechtigungen
- Konfigurieren Sie die Netzwerksicherheit

- Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole)
- Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI)
- Startpositionen für Abfragen und Streams
- Überwachen Ihrer Amazon-DocumentDB-Ereignisquelle
- Tutorial: Verwenden AWS Lambda mit Amazon DocumentDB Streams

Beispiel für Amazon-DocumentDB-Ereignis


```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
```

```
    ],
    "eventSource": "aws:docdb"
}
```

Weitere Informationen zu den Ereignissen in diesem Beispiel und den zugehörigen Formen finden Sie unter Änderungsereignisse auf der MongoDB-Dokumentationswebsite.

Voraussetzungen und Berechtigungen

Bevor Sie Amazon DocumentDB als Ereignisquelle für Ihre Lambda-Ereignisfunktion verwenden können, müssen Sie die folgenden Voraussetzungen beachten. Folgendes muss zutreffen:

- Haben Sie einen vorhandenen Amazon DocumentDB-Cluster in derselben AWS-Konto und AWS-Region wie Ihre Funktion. Wenn Sie über keinen bestehenden Cluster verfügen, können Sie anhand der Anleitung in Erste Schritte mit Amazon DocumentDB im Amazon-DocumentDB-Entwicklerhandbuch einen erstellen. Alternativ dazu führen Sie die ersten Schritte in Tutorial: Verwenden AWS Lambda mit Amazon DocumentDB Streams durch die Erstellung eines Amazon DocumentDB-Clusters mit allen erforderlichen Voraussetzungen.
- Sie müssen Lambda den Zugriff auf die Ressourcen von Amazon Virtual Private Cloud (Amazon VPC) erlauben, die Ihrem Amazon-DocumentDB-Cluster zugeordnet sind. Weitere Informationen finden Sie unter Konfigurieren Sie die Netzwerksicherheit.
- Aktivieren Sie TLS in Ihrem Amazon-DocumentDB-Cluster. Dies ist die Standardeinstellung. Wenn Sie TLS deaktivieren, kann Lambda nicht mit Ihrem Cluster kommunizieren.
- Aktivieren Sie Änderungsstreams in Ihrem Amazon-DocumentDB-Cluster. Weitere Informationen finden Sie unter Verwenden von Änderungsstreams mit Amazon DocumentDB im Amazon-DocumentDB-Entwicklerhandbuch.
- Stellen Sie Lambda Anmeldeinformationen für den Zugriff auf Ihren Amazon-DocumentDB-Cluster zur Verfügung. Geben Sie bei der Einrichtung der Ereignisquelle den AWS Secrets Manager-Schlüssel mit den Authentifizierungsdetails (Benutzername und Passwort) an, die für den Zugriff auf Ihren Cluster erforderlich sind. Um diesen Schlüssel während der Einrichtung bereitzustellen, haben Sie die folgenden Möglichkeiten:
  - Wenn Sie die Lambda-Konsole für die Einrichtung verwenden, geben Sie diesen Schlüssel in das Feld Secrets-Manager-Schlüssel ein.
  - Wenn Sie das AWS Command Line Interface (AWS CLI) für die Einrichtung verwenden, geben Sie diesen Schlüssel in der `source-access-configurations` Option an. Sie können diese Option entweder in den `create-event-source-mapping`-Befehl oder in den `update-event-source-mapping`-Befehl einschließen. Beispiel:

```
aws lambda create-event-source-mapping \
    ...
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
    ...
```

- Sie müssen Lambda Berechtigungen zur Verwaltung von Ressourcen gewähren, die zu Ihrem Amazon-DocumentDB-Stream gehören. Fügen Sie der zu der Ausführungsrolle Ihrer Funktion manuell die folgenden Berechtigungen hinzu:
  - rds:DescribeDBClusters
  - rds: DescribeDBClusterParameters
  - rds: DescribeDBSubnetGroups
  - ec2: CreateNetworkInterface
  - ec2: DescribeNetworkInterfaces
  - ec2: DescribeVpcs
  - ec2: DeleteNetworkInterface
  - ec2: DescribeSubnets
  - ec2: DescribeSecurityGroups
  - kms:Decrypt
  - Verwalter von Geheimnissen: GetSecretValue
- Halten Sie die Größe der Amazon-DocumentDB-Änderungsstream-Ereignisse, die Sie an Lambda senden, bei unter 6 MB. Lambda unterstützt Nutzlasten mit einer Größe von bis zu 6 MB. Wenn Ihr Änderungsstream versucht, Lambda ein Ereignis zu senden, das größer als 6 MB ist, löscht Lambda die Nachricht und gibt die `OversizedRecordCount`-Metrik aus. Lambda gibt alle Metriken auf die bestmögliche Weise aus.

> **Note**
>
> Während Lambda-Ereignisfunktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB und Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten. Diese

Einschränkung stellt sicher, dass die Ereignisquellenzuordnung Funktionsfehler und Wiederholungsversuche ordnungsgemäß verarbeiten kann.

Konfigurieren Sie die Netzwerksicherheit

Um Lambda vollen Zugriff auf Amazon DocumentDB über Ihre Zuordnung von Ereignisquellen zu ermöglichen, muss Ihr Cluster entweder einen öffentlichen Endpunkt (öffentliche IP-Adresse) verwenden oder Sie müssen Zugriff auf die Amazon VPC gewähren, in der Sie den Cluster erstellt haben.

Wenn Sie Amazon DocumentDB mit Lambda verwenden, erstellen Sie AWS PrivateLink VPC-Endpunkte, die Ihrer Event Function Zugriff auf die Ressourcen in Ihrer Amazon VPC gewähren.

> **Note**
>
> AWS PrivateLink VPC-Endpunkte sind für Funktionen mit Ereignisquellenzuordnungen erforderlich, die den Standardmodus (auf Abruf) für Ereignisabfragen verwenden. Wenn Ihre Ereignisquellenzuordnung den Bereitstellungsmodus verwendet, müssen Sie keine AWS PrivateLink VPC-Endpunkte konfigurieren.

Erstellen Sie einen Endpunkt, der den Zugriff auf die folgenden Ressourcen ermöglicht:

- Lambda – Erstellen Sie einen Endpunkt für den Lambda-Serviceprinzipal.
- AWS STS — Erstellen Sie einen Endpunkt für den, damit ein Service Principal eine Rolle AWS STS in Ihrem Namen übernehmen kann.
- Secrets Manager – Wenn Ihr Cluster Secrets Manager zum Speichern von Anmeldeinformationen verwendet, erstellen Sie einen Endpunkt für Secrets Manager.

Alternativ konfigurieren Sie ein NAT-Gateway auf jedem öffentlichen Subnetz in der Amazon-VPC. Weitere Informationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn Sie eine Zuordnung von Ereignisquellen für Amazon DocumentDB erstellen, prüft Lambda, ob Elastic-Network-Schnittstelle (ENI) für die für Ihre Amazon VPC konfigurierten Subnetze und Sicherheitsgruppen bereits vorhanden sind. Findet Lambda vorhandene ENIs, versucht es, diese wiederzuverwenden. Andernfalls erstellt Lambda neue ENIs, um eine Verbindung mit der Ereignisquelle herzustellen und Ihre Funktion aufzurufen.

> **Note**
>
> Lambda-Ereignisfunktionen werden immer in VPCs ausgeführt, die dem Lambda-Service gehören. Die VPC-Konfiguration Ihrer Funktion hat keinen Einfluss auf die Zuordnung von Ereignisquellen. Nur die Netzwerkkonfiguration der Ereignisquelle bestimmt, wie Lambda sich mit Ihrer Ereignisquelle verbindet.

Konfigurieren Sie die Sicherheitsgruppen für die Amazon VPC, die Ihren Cluster enthält. Standardmäßig verwendet Amazon DocumentDB die folgenden Ports: `27017`.

- Eingehende Regeln – Erlauben Sie den gesamten Datenverkehr auf dem Standard-Broker-Port für die Sicherheitsgruppe, die mit Ihrer Ereignisquelle verbunden ist. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr — Erlauben Sie den gesamten Datenverkehr auf dem Port `443` für externe Ziele, wenn Ihre Funktion mit Diensten kommunizieren muss. AWS Alternativ können Sie auch eine selbstreferenzierende Sicherheitsgruppenregel verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen Diensten kommunizieren müssen. AWS
- Regeln für eingehenden Datenverkehr auf Amazon-VPC-Endpunkten – Wenn Sie einen Amazon-VPC-Endpunkt verwenden, muss die Sicherheitsgruppe, die mit Ihrem Amazon-VPC-Endpunkt verbunden ist, eingehenden Verkehr auf Port `443` von der Cluster-Sicherheitsgruppe zulassen.

Wenn Ihr Cluster eine Authentifizierung verwendet, können Sie auch die Endpunktrichtlinie für den Secrets-Manager-Endpunkt einschränken. Für den Aufruf der Secrets-Manager-API verwendet Lambda Ihre Funktionsrolle und nicht den Lambda-Serviceprinzipal.

Example VPC-Endpunktrichtlinie – Secrets Manager-Endpunkt

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
```

```
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

Wenn Sie Amazon VPC-Endpunkte verwenden, AWS leitet Ihre API-Aufrufe zum Aufrufen Ihrer Funktion über das Elastic Network Interface (ENI) des Endpunkts weiter. Der Prinzipal des Lambda-Dienstes muss `lambda:InvokeFunction` für alle Rollen und Funktionen aufrufen, die diese ENI verwenden.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Um sicherzustellen, dass Ihre Ereignisquellenzuordnung Ihre Lambda-Ereignisfunktion aufrufen kann, muss die VPC-Endpunktrichtlinie dem Lambda-Serviceprinzipal erlauben, und aufzurufen. `sts:AssumeRole` `lambda:InvokeFunction` Wenn Sie Ihre VPC-Endpunktrichtlinien so einschränken, dass nur API-Aufrufe zugelassen werden, die aus Ihrem Unternehmen stammen, kann die Zuordnung von Ereignisquellen nicht richtig funktionieren, weshalb in diesen Richtlinien `"Resource": "*"` erforderlich ist.

Die folgenden Beispiel-VPC-Endpunktrichtlinien zeigen, wie der erforderliche Zugriff auf den Lambda-Serviceprinzipal für die AWS STS - und Lambda-Endpunkte gewährt wird.

Example VPC-Endpunktrichtlinie — AWS STS Endpunkt

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
```

```
    }
```

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole)

Damit eine Lambda-Event-Funktion aus dem Change-Stream eines Amazon DocumentDB-Clusters lesen kann, erstellen Sie eine Ereignisquellenzuordnung. In diesem Abschnitt wird die entsprechende Vorgehensweise über die Lambda-Konsole beschrieben. AWS SDK und AWS CLI Anweisungen finden Sie unter. the section called “Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI)”

So erstellen Sie eine Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie unter Auslöserkonfiguration in der Dropdown-Liste die Option DocumentDB aus.
5. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Optionen für Amazon-DocumentDB-Ereignisquellen:

- DocumentDB-Cluster – Wählen Sie einen Amazon-DocumentDB-Cluster aus.

- Auslöser aktivieren – Wählen Sie aus, ob Sie den Auslöser sofort aktivieren möchten. Wenn Sie dieses Kontrollkästchen aktivieren, beginnt Ihre Funktion sofort, Datenverkehr aus dem angegebenen Amazon-DocumentDB-Änderungsstream zu empfangen, sobald die Zuordnung von Ereignisquellen erstellt wurde. Wir empfehlen, das Kontrollkästchen zu deaktivieren, um die Zuordnung von Ereignisquellen zu Testzwecken in einem deaktivierten Zustand zu erstellen. Nach der Erstellung können Sie die Zuordnung von Ereignisquellen jederzeit aktivieren.
- Datenbankname – Geben Sie den Namen einer Datenbank innerhalb des Clusters ein, die verwendet werden soll.
- (Optional) Sammlungsname – Geben Sie den Namen einer Sammlung in der Datenbank ein, die verwendet werden soll. Wenn Sie keine Sammlung angeben, hört Lambda auf alle Ereignisse aus jeder Sammlung in der Datenbank.
- Batchgröße – Legen Sie die maximale Anzahl von Nachrichten fest, die in einem einzelnen Batch abgerufen werden sollen bis zu 10 000. Die Standardgröße beträgt 100.
- Startposition – Wählen Sie die Position im Stream aus, von der aus das Lesen von Datensätzen beginnen.
  - Neueste – Verarbeiten Sie Datensätze, die neu zum Stream hinzugefügt wurden. Ihre Funktion beginnt erst mit der Verarbeitung von Datensätzen, nachdem Lambda die Erstellung Ihrer Ereignisquelle abgeschlossen hat. Das bedeutet, dass einige Datensätze möglicherweise gelöscht werden, bis Ihre Ereignisquelle erfolgreich erstellt wurde.
  - Horizont trimmen – Verarbeiten Sie alle Datensätze im Stream. Lambda verwendet die Protokollspeicherdauer Ihres Clusters, um zu bestimmen, ab wo mit dem Lesen von Ereignissen begonnen werden soll. Insbesondere beginnt Lambda mit dem Lesen von `current_time - log_retention_duration`. Ihr Änderungsstream muss bereits vor diesem Zeitstempel aktiv sein, damit Lambda alle Ereignisse korrekt lesen kann.
  - Am Zeitstempel – Verarbeiten Sie Datensätze ab einem bestimmten Zeitpunkt. Ihr Änderungsstream muss bereits vor dem angegebenen Zeitstempel aktiv sein, damit Lambda alle Ereignisse korrekt lesen kann.
- Authentifizierung – Wählen Sie unter Authentifizierungsmethode den Zugriff auf den Broker in Ihrem Cluster aus.
  - BASIC_AUTH – Bei der Standardauthentifizierung müssen Sie den Secrets-Manager-Schlüssel angeben, der die Anmeldeinformationen für den Zugriff auf Ihren Cluster enthält.
- Secrets-Manager-Schlüssel – Wählen Sie den Secrets-Manager-Schlüssel aus, der die Authentifizierungsdetails (Benutzername und Passwort) enthält, die für den Zugriff auf Ihren Amazon-DocumentDB-Cluster erforderlich sind.

- (Optional) Batch-Fenster – Legen Sie die maximale Zeitspanne zur Erfassung von Datensätzen vor dem Aufruf der Funktion in Sekunden fest (bis zu 300).
- (Optional) Vollständige Dokumentkonfiguration – Wählen Sie für Dokumentaktualisierungsvorgänge aus, was Sie an den Stream senden möchten. Der Standardwert ist `Default`, was bedeutet, dass Amazon DocumentDB für jedes Änderungsstream-Ereignis nur ein Delta sendet, das die vorgenommenen Änderungen beschreibt. Weitere Informationen zu diesem Feld finden Sie FullDocumentin der MongoDB Javadoc-API-Dokumentation.
  - Standard – Lambda sendet nur ein Teildokument, das die vorgenommenen Änderungen beschreibt.
  - UpdateLookup— Lambda sendet ein Delta, das die Änderungen beschreibt, zusammen mit einer Kopie des gesamten Dokuments.

Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI)

Um eine Zuordnung von Ereignisquellen von Amazon DocumentDB mit einem AWS SDK zu verwalten, können Sie die folgenden API-Operationen verwenden:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Verwenden Sie den `create-event-source-mapping`Befehl, um die Zuordnung der AWS CLI Ereignisquelle mit dem zu erstellen. Im folgenden Beispiel wird dieser Befehl verwendet, um eine Funktion namens `my-function` einem Amazon-DocumentDB-Änderungsstream zuzuordnen. Der Ereignisquelle wird mit dem Amazon-Ressourcennamen (ARN) angegeben, mit einer Batchgröße von 500, beginnend ab dem Zeitstempel in Unix-Zeit. Der Befehl gibt auch den Secrets-Manager-Schlüssel an, den Lambda verwendet, um eine Verbindung zu Amazon DocumentDB herzustellen. Darüber hinaus enthält er `document-db-event-source-config`-Parameter, die die Datenbank und die Sammlung angeben, aus der gelesen werden soll.

```
aws lambda create-event-source-mapping --function-name my-function \
```

```
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

Nach der Erstellung können Sie den `update-event-source-mapping`-Befehl verwenden, um die Einstellungen für Ihre Amazon-DocumentDB-Ereignisquelle zu aktualisieren. Im folgenden Beispiel werden die Batchgröße auf 1 000 und das Batchfenster auf 10 Sekunden aktualisiert. Für diesen Befehl benötigen Sie die UUID Ihrer Zuordnung von Ereignisquellen, die Sie über den `list-event-source-mapping`-Befehl oder die Lambda-Konsole abrufen können.

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda aktualisiert die Einstellungen asynchron, sodass Sie diese Änderungen möglicherweise erst in der Ausgabe sehen, wenn der Vorgang abgeschlossen ist. Verwenden Sie den `get-event-source-mapping`-Befehl, um die aktuellen Einstellungen Ihrer Zuordnung von Ereignisquellen anzuzeigen.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
    "MaximumBatchingWindowInSeconds": 10,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
```

```
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

Um Ihre Zuordnung von Ereignisquellen von Amazon DocumentDB zu löschen, verwenden Sie den `delete-event-source-mapping`-Befehl.

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

Startpositionen für Abfragen und Streams

Beachten Sie, dass die Stream-Abfrage bei der Erstellung und Aktualisierung der Zuordnung von Ereignisquellen letztendlich konsistent ist.

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Bei Aktualisierungen der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass, wenn Sie `LATEST` als Startposition für den Stream angeben, die Zuordnung von Ereignisquellen bei der Erstellung oder Aktualisierung möglicherweise Ereignisse übersieht. Um sicherzustellen, dass keine Ereignisse übersehen werden, geben Sie die Startposition des Streams als `TRIM_HORIZON` oder `AT_TIMESTAMP` an.

Überwachen Ihrer Amazon-DocumentDB-Ereignisquelle

Um Sie bei der Überwachung Ihrer Amazon-DocumentDB-Ereignisquelle zu unterstützen, gibt Lambda die Metrik `IteratorAge` aus, wenn die Funktion das Verarbeiten von Datensätzen abgeschlossen hat. Das Iterator-Alter ist der Unterschied zwischen dem Zeitstempel des letzten Ereignisses und dem aktuellen Zeitstempel. Im Wesentlichen gibt die Metrik `IteratorAge` an, wie alt der letzte Datensatz im Batch bei Fertigstellung der Verarbeitung war.

Wenn Ihre Funktion derzeit neue Ereignisse verarbeitet, können Sie mit dem Iterator-Alter die Latenz zwischen dem Zeitpunkt, zu dem ein Datensatz hinzugefügt wird, und dem Zeitpunkt, zu dem die Funktion verarbeitet wird, schätzen. Eine ansteigende Tendenz bei `IteratorAge` kann auf Probleme mit Ihrer Funktion hindeuten. Weitere Informationen finden Sie unter CloudWatch Metriken mit Lambda verwenden.

Amazon DocumentDB-Change-Streams sind nicht für die Bewältigung großer Zeitlücken zwischen Ereignissen optimiert. Wenn Ihre Amazon DocumentDB DocumentDB-Ereignisquelle über einen längeren Zeitraum keine Ereignisse empfängt, deaktiviert Lambda möglicherweise die Zuordnung der Ereignisquelle. Die Dauer dieses Zeitraums kann je nach Größe des Clusters und anderen Workloads zwischen einigen Wochen und einigen Monaten variieren.

Lambda unterstützt Nutzlasten von bis zu 6 MB. Änderungsstream-Ereignisse von Amazon DocumentDB können jedoch bis zu 16 MB groß sein. Wenn Ihr Änderungsstream versucht, Lambda ein Änderungsstream-Ereignis zu senden, das größer als 6 MB ist, löscht Lambda die Nachricht und gibt die Metrik `OversizedRecordCount` aus. Lambda gibt alle Metriken auf die bestmögliche Weise aus.

Tutorial: Verwenden AWS Lambda mit Amazon DocumentDB Streams

In diesem Tutorial erstellen Sie eine grundlegende Lambda-Ereignisfunktion, die Ereignisse aus einem Amazon DocumentDB DocumentDB-Change-Stream (mit MongoDB-Kompatibilität) verarbeitet. Um dieses Tutorial abzuschließen, durchlaufen Sie die folgenden Phasen:

- Richten Sie Ihren Amazon-DocumentDB-Cluster ein, stellen Sie eine Verbindung zu ihm her und aktivieren Sie Change-Streams darauf.
- Erstellen Sie Ihre Lambda-Ereignisfunktion und konfigurieren Sie Ihren Amazon DocumentDB-Cluster als Ereignisquelle für Ihre Funktion.
- Testen Sie die Einrichtung, indem Sie Elemente in Ihre Amazon-DocumentDB-Datenbank einfügen.

Erstellen Sie den Amazon DocumentDB-Cluster

1. Öffnen Sie die Amazon DocumentDB-Konsole. Wählen Sie unter Cluster die Option Erstellen aus.
2. Erstellen Sie einen Cluster mit der folgenden Konfiguration:
   - Wählen Sie als Clustertyp die Option Instance-based Cluster aus. Dies ist die Standardoption.
   - Stellen Sie sicher, dass unter Clusterkonfiguration die Engine-Version 5.0.0 ausgewählt ist. Dies ist die Standardoption.
   - Unter Instance-Konfiguration:
     - Wählen Sie für DB-Instance-Klasse die Option Speicheroptimierte Klassen aus. Dies ist die Standardoption.
     - Wählen Sie für Anzahl der regulären Replikat-Instances den Wert 1 aus.

- Verwenden Sie für Instance-Klasse die Standardauswahl.
- Geben Sie unter Authentifizierung einen Benutzernamen für den Hauptbenutzer ein und wählen Sie dann Selbstverwaltet aus. Geben Sie ein Passwort ein und bestätigen Sie es.
- Behalten Sie die alle Standardeinstellung bei.

3. Wählen Sie Cluster erstellen.

Erstellen des Secrets in Secrets Manager

Während Amazon DocumentDB Ihren Cluster erstellt, erstellen Sie ein AWS Secrets Manager Geheimnis, um Ihre Datenbankanmeldedaten zu speichern. Sie geben diesen geheimen Schlüssel an, wenn Sie in einem späteren Schritt die Zuordnung von Ereignisquellen in Lambda erstellen.

Um das Secret in Secrets Manager zu erstellen

1. Öffnen Sie die Secrets-Manager-Konsole und wählen Sie Neues Secret speichern.
2. Wählen Sie für Secret-Typ auswählen eine der folgenden Optionen aus:
   - Unter Grundlegende Details:
     - Secret-Typ: Anmeldeinformationen für die Amazon-DocumentDB-Datenbank
     - Geben Sie unter Anmeldeinformationen den Benutzernamen und das Passwort ein, den bzw. das Sie zum Erstellen Ihres Amazon-DocumentDB-Clusters verwendet haben.
     - Datenbank: Wählen Sie Ihren Amazon DocumentDB-Cluster aus.
     - Wählen Sie Weiter aus.
3. Wählen Sie unter Secret konfigurieen eine der folgenden Optionen aus:
   - Geheimer Name: `DocumentDBSecret`
   - Wählen Sie Weiter aus.
4. Wählen Sie Weiter aus.
5. Wählen Sie Store (Speichern) aus.
6. Aktualisieren Sie die Konsole, um zu überprüfen, ob Sie das `DocumentDBSecret`-Secret erfolgreich gespeichert haben.

Notieren Sie sich den geheimen ARN. Sie werden es in einem späteren Schritt benötigen.

Verbinden mit dem Cluster

Connect zu Ihrem Amazon DocumentDB-Cluster her mit AWS CloudShell

1. Suchen Sie in der Amazon-DocumentDB-Verwaltungskonsole unter Cluster den von Ihnen erstellten Cluster. Wählen Sie Ihren Cluster aus, indem Sie das Kontrollkästchen neben dem Cluster aktivieren.
2. Wählen Sie Mit Cluster verbinden aus. Der Befehlsbildschirm CloudShell Ausführen wird angezeigt.
3. Geben Sie im Feld Neuer Umgebungsname einen eindeutigen Namen ein, z. B. „Test“, und wählen Sie Erstellen und ausführen aus.
4. Geben Sie bei der Aufforderung Ihr Passwort ein. Wenn der Prompt zu `rs0 [direct: primary] <env-name>>` wechselt, sind Sie erfolgreich mit Ihrem Amazon-DocumentDB-Cluster verbunden.

Change-Streams aktivieren

In diesem Tutorial verfolgen Sie Änderungen an der `products` Sammlung der `docdbdemo` Datenbank in Ihrem Amazon DocumentDB-Cluster. Sie tun dies, indem Sie Change-Streams aktivieren.

Um eine neue Datenbank in Ihrem Cluster zu erstellen

1. Führen Sie den folgenden Befehl aus, um eine neue Datenbank mit dem Namen `docdbdemo` zu erstellen:

   ```
   use docdbdemo
   ```

2. Geben Sie im Terminalfenster den folgenden Befehl ein, um einen Datensatz in `docdbdemo` einzufügen:

   ```
   db.products.insertOne({"hello":"world"})
   ```

   Sie sollten eine Ausgabe wie die Folgende sehen:

   ```
   {
     acknowledged: true,
     insertedId: ObjectId('67f85066ca526410fd531d59')
   ```

```
}
```

3. Dann aktivieren Sie mit dem folgenden Befehl Change-Streams für die `products`-Sammlung der `docdbdemo`-Datenbank:

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

Schnittstellen-VPC-Endpunkte erstellen

Erstellen Sie als Nächstes Schnittstellen-VPC-Endpunkte, um sicherzustellen, dass Lambda und Secrets Manager (der später zum Speichern unserer Cluster-Anmeldeinformationen verwendet wird) eine Verbindung zu Ihrer Standard-VPC herstellen können.

Um Schnittstellen-VPC-Endpunkte zu erstellen

1. Öffnen Sie die VPC;-Konsole. Wählen Sie im linken Menü unter Virtual Private Cloud die Option Endpunkte aus.
2. Wählen Sie Endpunkt erstellen aus. Erstellen Sie einen Endpunkt mit der folgenden Konfiguration:
   - Geben Sie für Name Tag `lambda-default-vpc` ein.
   - Wählen Sie als Servicekategorie die Option AWS Dienste aus.
   - Geben Sie unter Services `lambda` in das Suchfeld ein. Wählen Sie den Service im Format `com.amazonaws.<region>.lambda`.
   - Wählen Sie für VPC die VPC aus, in der sich Ihr Amazon-DocumentDB-Cluster befindet. Dies ist in der Regel die Standard-VPC.
   - Aktivieren Sie für Subnetze die Kästchen neben jeder Availability Zone. Wählen Sie die richtige Subnetz-ID für jede Availability Zone.
   - Wählen Sie unter IP-Adresstyp IPv4 aus.

- Wählen Sie für Sicherheitsgruppen die Sicherheitsgruppe aus, die Ihr Amazon-DocumentDB-Cluster verwendet. Dies ist in der Regel die `default`-Sicherheitsgruppe.
- Behalten Sie die alle Standardeinstellung bei.
- Wählen Sie Endpunkt erstellen aus.

3. Wählen Sie erneut Endpunkt erstellen. Erstellen Sie einen Endpunkt mit der folgenden Konfiguration:

   - Geben Sie für Name Tag `secretsmanager-default-vpc` ein.
   - Wählen Sie als Servicekategorie die Option AWS Dienste aus.
   - Geben Sie unter Services `secretsmanager` in das Suchfeld ein. Wählen Sie den Service im Format `com.amazonaws.<region>.secretsmanager`.
   - Wählen Sie für VPC die VPC aus, in der sich Ihr Amazon-DocumentDB-Cluster befindet. Dies ist in der Regel die Standard-VPC.
   - Aktivieren Sie für Subnetze die Kästchen neben jeder Availability Zone. Wählen Sie die richtige Subnetz-ID für jede Availability Zone.
   - Wählen Sie unter IP-Adresstyp IPv4 aus.
   - Wählen Sie für Sicherheitsgruppen die Sicherheitsgruppe aus, die Ihr Amazon-DocumentDB-Cluster verwendet. Dies ist in der Regel die `default`-Sicherheitsgruppe.
   - Behalten Sie die alle Standardeinstellung bei.
   - Wählen Sie Endpunkt erstellen aus.

Damit ist der Abschnitt zur Einrichtung eines Clusters in diesem Tutorial abgeschlossen.

Erstellen der Ausführungsrolle

In den nächsten Schritten erstellen Sie Ihre Lambda-Event-Funktion. Zunächst müssen Sie die Ausführungsrolle erstellen, die Ihrer Funktion die Berechtigung zum Zugriff auf Ihren Cluster gibt. Dazu erstellen Sie zunächst eine IAM-Richtlinie und verknüpfen diese dann mit einer IAM-Rolle.

So erstellen Sie eine IAM-Richtlinie

1. Öffnen Sie in der IAM-Konsole die Seite Richtlinien und wählen Sie dann Richtlinie erstellen aus.
2. Wählen Sie den Tab JSON. Ersetzen Sie in der folgenden Richtlinie den ARN der Secrets-Manager-Ressource in der letzten Zeile der Anweisung durch Ihren geheimen ARN von zuvor, und kopieren Sie die Richtlinie in den Editor.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

3. Wählen Sie Weiter: Tags und dann Weiter: Prüfen aus.

4. Geben Sie unter Name `AWSDocumentDBLambdaPolicy` ein.
5. Wählen Sie Richtlinie erstellen aus.

So erstellen Sie die IAM-Rolle

1. Öffnen Sie die Seite Rollen in der IAM-Konsole und wählen Sie Rolle erstellen.
2. Wählen Sie für Vertrauenswürdige Entität auswählen die folgenden Optionen:
   - Typ der vertrauenswürdigen Entität: Dienst AWS
   - Service oder Anwendungsfall: Lambda
   - Wählen Sie Weiter aus.
3. Wählen Sie für Berechtigungen hinzufügen die `AWSDocumentDBLambdaPolicy` Richtlinie aus, die Sie gerade erstellt haben, sowie die, `AWSLambdaBasicExecutionRole` um Ihrer Funktion Berechtigungen zum Schreiben in Amazon CloudWatch Logs zu erteilen.
4. Wählen Sie Weiter aus.
5. Geben Sie für Role name (Rollenname) den Namen `AWSDocumentDBLambdaExecutionRole` ein.
6. Wählen Sie Rolle erstellen aus.

Erstellen Sie die Lambda-Ereignisfunktion

Dieses Tutorial verwendet die Python 3.14-Laufzeit, aber wir haben auch Beispielcodedateien für andere Laufzeiten bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um den Code für die Laufzeit zu sehen, an der Sie interessiert sind.

Der Code empfängt eine Amazon-DocumentDB-Ereigniseingabe und verarbeitet die darin enthaltene Nachricht.

So erstellen Sie die Lambda-Ereignisfunktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Wählen Sie Ohne Vorgabe erstellen aus.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie unter Funktionsname `ProcessDocumentDBRecords` ein.

b. Wählen Sie für Runtime Python 3.14.

c. Wählen Sie für Architektur x86_64 aus.

5. Gehen Sie auf der Registerkarte Standard-Ausführungsrolle ändern wie folgt vor:

a. Erweitern Sie die Registerkarte und wählen Sie dann Verwenden einer vorhandenen Rolle aus.

b. Wählen Sie die zuvor erstellte `AWSDocumentDBLambdaExecutionRole` aus.

6. Wählen Sie „Ereignisfunktion erstellen“.

Um den Code für die Ereignisfunktion bereitzustellen

1. Wählen Sie im folgenden Feld die Registerkarte Python und kopieren Sie den Code.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von .NET.

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace LambdaDocDb;

public class Function
```

```
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
 ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument,
 new JsonSerializerOptions { WriteIndented = true });

        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
```

```
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }

        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }
```

```
    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Go.

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
```

```
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Java.

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }
```

```
    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");

        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Verwenden eines Amazon DocumentDB DocumentDB-Ereignisses mit Lambda.
JavaScript

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};
```

```
const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

Ein Amazon DocumentDB DocumentDB-Ereignis mit Lambda verwenden TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from
 'aws-lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details
```

```
        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument,
 JSON_PRETTY_PRINT) . "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Python.

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
```

```
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Ruby.

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

SDK für Rust

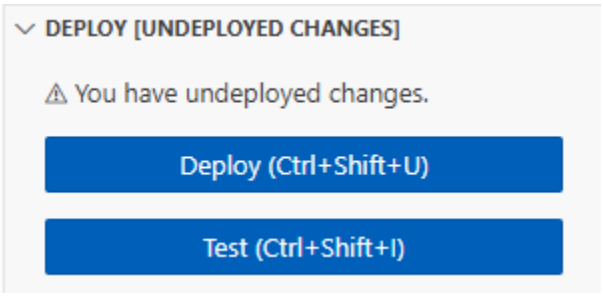
> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features
 = ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

2. Fügen Sie den Code im Code-Quellbereich der Lambda-Konsole in den Code-Editor ein und ersetzen Sie dabei den von Lambda erstellten Code.
3. Wählen Sie im Abschnitt DEPLOY die Option Deploy aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen Sie die Zuordnung von Ereignisquellen in Lambda

Erstellen Sie die Ereignisquellenzuordnung, die Ihren Amazon DocumentDB DocumentDB-Change-Stream mit Ihrer Lambda-Ereignisfunktion verknüpft. Nachdem Sie diese Ereignisquellenzuordnung erstellt haben, beginnt AWS Lambda sofort die Abfrage des Streams.

Die Zuordnung von Ereignisquellen erstellen

1. Öffnen Sie die Seite Event Functions in der Lambda-Konsole.
2. Wählen Sie die `ProcessDocumentDBRecords`-Funktion aus, die Sie zuvor erstellt haben.
3. Wählen Sie die Registerkarte Konfiguration und dann im linken Menü die Option Auslöser aus.
4. Wählen Sie Add trigger.
5. Wählen Sie unter Trigger-Konfiguration als Quelle die Option Amazon DocumentDB aus.
6. Erstellen Sie die Zuordnung von Ereignisquellen mit der folgenden Konfiguration:
   - Amazon-DocumentDB-Cluster: Wählen Sie den Cluster aus, den Sie zuvor erstellt haben.
   - Datenbankname: docdbdemo
   - Name der Kollektion:Produkte
   - Chargengröße: 1
   - Startposition: Neueste
   - Authentifizierung: BASIC_AUTH
   - Secrets-Manager-Schlüssel: Wählen Sie das Secret für Ihren Amazon-DocumentDB-Cluster aus. Es heißt so etwas wie `rds!cluster-12345678-a6f0-52c0-b290-db4aga89274f`.
   - Batchfenster: 1
   - Vollständige Konfiguration des Dokuments: UpdateLookup
7. Wählen Sie Hinzufügen aus. Die Erstellung Ihrer Zuordnung von Ereignisquellen kann einige Minuten dauern.

Testen Sie Ihre Event-Funktion

Warten Sie, bis die Zuordnung von Ereignisquellen den Status Aktiviert erreicht hat. Dies kann mehrere Minuten dauern. Testen Sie anschließend die End-to-End-Konfiguration, indem Sie Datenbankdatensätze einfügen, aktualisieren und löschen. Bevor Sie beginnen:

1. Stellen Sie erneut eine Verbindung zu Ihrem Amazon DocumentDB-Cluster in Ihrer CloudShell Umgebung her.
2. Führen Sie den folgenden Befehl aus, um sicherzustellen, dass Sie die `docdbdemo` Datenbank verwenden:

```
use docdbdemo
```

Einfügen eines Datensatzes

Fügen Sie einen Datensatz in die `products`-Sammlung der `docdbdemo`-Datenbank ein:

```
db.products.insertOne({"name":"Pencil", "price": 1.00})
```

Überprüfen Sie anhand von CloudWatch Logs, ob Ihre Funktion dieses Ereignis erfolgreich verarbeitet hat. Sie sollten einen Protokolleintrag wie diesen sehen:

Timestamp	Message
2025-05-05T23:55:23.474Z	Operation type: insert
2025-05-05T23:55:23.474Z	db: docdbdemo
2025-05-05T23:55:23.474Z	collection: products
2025-05-05T23:55:23.474Z	Full document: {
2025-05-05T23:55:23.474Z	"_id": {
2025-05-05T23:55:23.474Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:23.474Z	},
2025-05-05T23:55:23.474Z	"name": "Pencil",
2025-05-05T23:55:23.474Z	"price": 1
2025-05-05T23:55:23.474Z	}

Aktualisieren von Datensätzen

Aktualisieren Sie den gerade eingefügten Datensatz mit dem folgenden Befehl:

```
db.products.updateOne(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

Überprüfen Sie, ob Ihre Funktion dieses Ereignis erfolgreich verarbeitet hat, indem Sie die Option CloudWatch Logs überprüfen. Sie sollten einen Protokolleintrag wie diesen sehen:

Timestamp	Message
2025-05-05T23:55:27.918Z	Operation type: update
2025-05-05T23:55:27.918Z	db: docdbdemo
2025-05-05T23:55:27.918Z	collection: products
2025-05-05T23:55:27.918Z	Full document: {
2025-05-05T23:55:27.918Z	"_id": {
2025-05-05T23:55:27.918Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:27.918Z	},
2025-05-05T23:55:27.918Z	"name": "Pencil",
2025-05-05T23:55:27.918Z	"price": 0.5
2025-05-05T23:55:27.918Z	}

Einen Datensatz löschen

Löschen Sie den Datensatz, den Sie gerade aktualisiert haben, mit dem folgenden Befehl:

```
db.products.deleteOne( { "name": "Pencil" } )
```

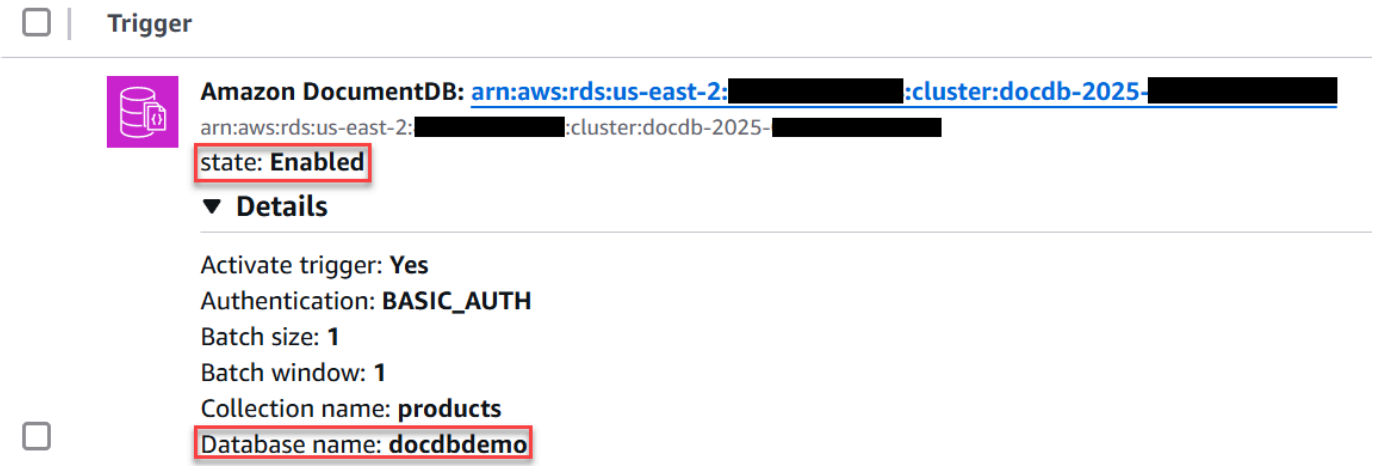
Überprüfen Sie anhand von CloudWatch Logs, ob Ihre Funktion dieses Ereignis erfolgreich verarbeitet hat. Sie sollten einen Protokolleintrag wie diesen sehen:

Timestamp	Message
2025-05-05T23:55:32.362Z	Operation type: delete
2025-05-05T23:55:32.362Z	db: docdbdemo
2025-05-05T23:55:32.362Z	collection: products
2025-05-05T23:55:32.362Z	Full document: {}

Fehlerbehebung

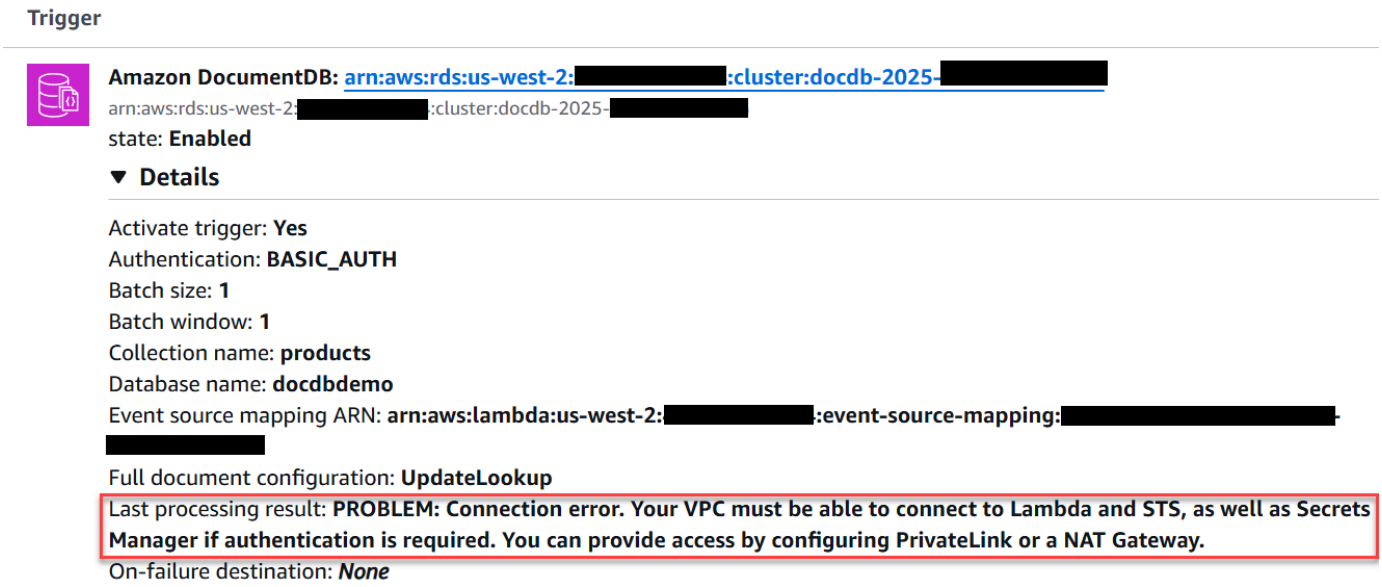
Wenn Sie in den CloudWatch Protokollen Ihrer Funktion keine Datenbankereignisse sehen, überprüfen Sie Folgendes:

- Stellen Sie sicher, dass sich die Zuordnung von Ereignisquellen in Lambda (auch als Auslöser bezeichnet) im Status Aktiviert befindet. Die Zuordnung von Ereignisquellen kann mehrere Minuten in Anspruch nehmen.
- Wenn die Zuordnung der Ereignisquellen aktiviert ist, Sie aber immer noch keine Datenbankereignisse sehen in CloudWatch:
  - Stellen Sie sicher, dass der Datenbankname in der Zuordnung von Ereignisquellen auf `docdbdemo` eingestellt ist.



  - Überprüfen Sie das Feld Letztes Verarbeitungsergebnis der Zuordnung von Ereignisquellen auf die folgende Meldung: „PROBLEM: Verbindungsfehler. Ihre VPC muss eine Verbindung zu Lambda und STS sowie zu Secrets Manager herstellen können, wenn eine Authentifizierung erforderlich ist.“ Wenn diese Fehlermeldung angezeigt wird, stellen Sie sicher, dass Sie die

Lambda- und Secrets-Manager-VPC-Schnittstellenendpunkte erstellt haben und dass die Endpunkte dieselbe VPC und dieselben Subnetze verwenden wie Ihr Amazon-DocumentDB-Cluster.



Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS -Ressourcen, die Sie nicht mehr verwenden, können Sie verhindern, dass unnötige Gebühren in Ihrem AWS-Konto-Konto anfallen.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.

4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Löschen von VPC-Endpunkten

1. Öffnen Sie die VPC;-Konsole. Wählen Sie im linken Menü unter Virtual Private Cloud die Option Endpunkte aus.
2. Wählen Sie die Endpunkte aus, die Sie erstellt haben.
3. Wählen Sie Actions (Aktionen), Delete VPC Endpoint (VPC-Endpunkte löschen).
4. Geben Sie **`delete`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

So löschen Sie den Amazon DocumentDB-Cluster

1. Öffnen Sie die Amazon DocumentDB-Konsole.
2. Wählen Sie den Amazon DocumentDB-Cluster aus, den Sie für dieses Tutorial erstellt haben, und deaktivieren Sie den Löschschutz.
3. Wählen Sie auf der Cluster-Hauptseite erneut Ihren Amazon DocumentDB-Cluster aus.
4. Wählen Sie Aktionen, Löschen aus.
5. Wählen Sie für Finalen Cluster-Snapshot erstellen die Option Nein aus.
6. Geben Sie **`delete`** in das Texteingabefeld ein.
7. Wählen Sie Löschen aus.

So löschen Sie das Secret im Secrets Manager

1. Öffnen Sie die Secrets Manager-Konsole.
2. Wählen Sie das Secret aus, das Sie für dieses Tutorial erstellt haben.
3. Wählen Sie Aktionen, Secret löschen.
4. Wählen Sie Schedule deletion.

Verwenden AWS Lambda mit Amazon DynamoDB

> **Note**
>
> Wenn Sie Daten an ein anderes Ziel als eine Lambda-Ereignisfunktion senden oder die Daten vor dem Senden anreichern möchten, finden Sie weitere Informationen unter Amazon EventBridge Pipes.

Sie können eine AWS Lambda Event-Funktion verwenden, um Datensätze in einem Amazon DynamoDB DynamoDB-Stream zu verarbeiten. Mit DynamoDB Streams können Sie eine Lambda-Ereignisfunktion auslösen, um bei jeder Aktualisierung einer DynamoDB-Tabelle zusätzliche Arbeit auszuführen.

Bei der Verarbeitung von DynamoDB-Streams müssen Sie eine Logik für teilweise Batch-Antworten implementieren, um zu verhindern, dass versucht wird, erfolgreich verarbeitete Datensätze erneut zu verarbeiten, wenn einige Datensätze in einem Batch fehlschlagen. Das Batch Processor Utility von Powertools for AWS Lambda ist in Python TypeScript, .NET und Java verfügbar und vereinfacht diese Implementierung, indem es automatisch die partielle Batch-Antwortlogik verarbeitet, wodurch die Entwicklungszeit reduziert und die Zuverlässigkeit verbessert wird.

Themen

- Abrufen und Stapeln von Streams
- Startpositionen für Abfragen und Streams
- Gleichzeitige Leser eines Shard in DynamoDB Streams
- Beispielereignis
- DynamoDB-Datensätze mit Lambda verarbeiten
- Konfigurieren einer teilweisen Batch-Antwort mit DynamoDB und Lambda
- Verworfene Datensätze für eine DynamoDB-Ereignisquelle in Lambda aufbewahren
- Implementierung der statusbehafteten DynamoDB-Stream-Verarbeitung in Lambda
- Lambdaparameter für Amazon DynamoDB-SQS-Zuordnungen von Ereignisquellen
- Verwendung der Ereignisfilterung mit einer DynamoDB-Ereignisquelle
- Tutorial: Verwenden AWS Lambda mit Amazon DynamoDB DynamoDB-Streams

Abrufen und Stapeln von Streams

Lambda fragt Shards in Ihrem DynamoDB-Stream nach Datensätzen bei einer Basisrate von viermal pro Sekunde ab. Sind Datensätze verfügbar, ruft Lambda Ihre Funktion auf und wartet auf das Ergebnis. Ist die Verarbeitung erfolgreich, setzt Lambda die Abrufe fort, bis es weitere Datensätze erhält.

Standardmäßig ruft Lambda Ihre Funktion auf, sobald Datensätze verfügbar sind. Wenn der Batch, den Lambda aus der Ereignisquelle liest, nur einen Datensatz enthält, sendet Lambda nur einen Datensatz an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle anweisen, Datensätze bis zu 5 Minuten lang zu puffern, indem Sie ein Batch-Fenster konfigurieren. Bevor die Funktion aufgerufen wird, liest Lambda so lange Datensätze aus der Ereignisquelle, bis es einen vollständigen Batch erfasst hat, das Batch-Verarbeitungsfenster abläuft oder der Batch die Nutzlastgrenze von 6 MB erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Lambda wartet mit dem Senden des nächsten zu verarbeitenden Stapels nicht, bis ggf. konfigurierte Erweiterungen abgeschlossen sind. Anders ausgedrückt: Ihre Erweiterungen werden möglicherweise weiter ausgeführt, während Lambda den nächsten Stapel von Datensätzen verarbeitet. Dies kann zu Drosselungsproblemen führen, wenn Sie gegen eine Einstellung oder gegen einen Grenzwert im Zusammenhang mit der Parallelität Ihres Kontos verstoßen. Um zu erkennen, ob möglicherweise ein Problem vorliegt, müssen Sie Ihre Funktionen überwachen sowie überprüfen, ob für Ihre Zuordnung von Ereignisquellen unerwartet hohe Parallelitätsmetriken vorliegen. Aufgrund der kurzen Zeit zwischen den Aufrufen kann Lambda kurzzeitig eine höhere Gleichzeitigkeitsnutzung als die Anzahl der Shards melden. Dies kann sogar für Lambda Event Functions ohne Erweiterungen gelten.

Konfigurieren Sie die ParallelizationFactorEinstellung so, dass ein Shard eines DynamoDB-Streams mit mehr als einem Lambda-Aufruf gleichzeitig verarbeitet wird. Sie können die Anzahl der gleichzeitigen Batches angeben, die Lambda von einem Shard abfragt, indem Sie

einen Parallelisierungsfaktor von 1 (Standard) bis 10 verwenden. Wenn Sie beispielsweise `ParallelizationFactor` auf 2 setzen, können Sie maximal 200 gleichzeitige Lambda-Aufrufe haben, um 100 DynamoDB-Stream-Shards zu verarbeiten (obwohl Sie in der Praxis möglicherweise unterschiedliche Werte für die Metrik sehen). `ConcurrentExecutions` Dies hilft, den Verarbeitungsdurchsatz hochzuskalieren, wenn das Datenvolumen flüchtig ist und IteratorAge hoch ist. Mit `ParallelizationFactor` führt Lambda mehrere gleichzeitige Batches pro Shard aus und gruppiert die Datensätze in jedem Shard nach Partitionsschlüssel. Innerhalb eines Shards werden Datensätze, die sich einen Partitionsschlüssel teilen, in denselben gleichzeitigen Batch verschoben und der Reihe nach verarbeitet. Datensätze mit unterschiedlichen Partitionsschlüsseln können gleichzeitig verarbeitet werden. DynamoDB Streams garantiert die Reihenfolge pro Element (Partitionsschlüssel und Sortierschlüssel). Daher werden die Stream-Datensätze für jedes einzelne Element immer in der Reihenfolge angezeigt, in der das Element geändert wurde.

Startpositionen für Abfragen und Streams

Beachten Sie, dass die Stream-Abfrage bei der Erstellung und Aktualisierung der Zuordnung von Ereignisquellen letztendlich konsistent ist.

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Bei Aktualisierungen der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass, wenn Sie `LATEST` als Startposition für den Stream angeben, die Zuordnung von Ereignisquellen bei der Erstellung oder Aktualisierung möglicherweise Ereignisse übersieht. Um sicherzustellen, dass keine Ereignisse übersehen werden, geben Sie die Stream-Startposition als `TRIM_HORIZON` an.

Gleichzeitige Leser eines Shard in DynamoDB Streams

Für Tabellen mit einer Region, bei denen es sich nicht um globale Tabellen handelt, können Sie festlegen, dass bis zu zwei Lambda-Ereignisfunktionen gleichzeitig aus demselben DynamoDB-Streams-Shard gelesen werden. Eine Überschreitung dieses Grenzwerts kann zu einer Anforderungsdrosselung führen. Für globale Tabellen empfehlen wir, die Anzahl der gleichzeitigen Funktionen auf eine zu beschränken, um eine Anforderungsdrosselung zu vermeiden.

Beispielereignis

Example


```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
```

```
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

DynamoDB-Datensätze mit Lambda verarbeiten

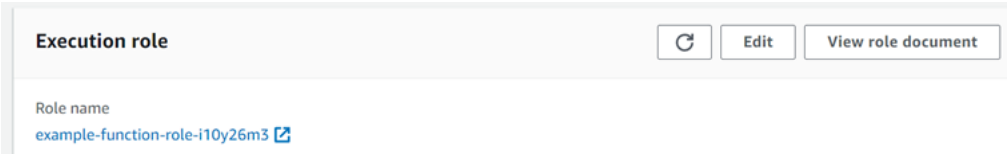
Erstellen Sie eine Ereignisquellenzuordnung, um Lambda anzuweisen, Datensätze aus Ihrem Stream an eine Lambda-Ereignisfunktion zu senden. Sie können mehrere Ereignisquellenzuordnungen erstellen, um dieselben Daten mit mehreren Lambda-Ereignisfunktionen zu verarbeiten oder um Elemente aus mehreren Streams mit einer einzigen Funktion zu verarbeiten.

Sie können Zuordnungen von Ereignisquellen konfigurieren, um Datensätze aus einem Stream in einem anderen AWS-Konto zu verarbeiten. Weitere Informationen hierzu finden Sie unter the section called “Cross-account Zuordnungen”.

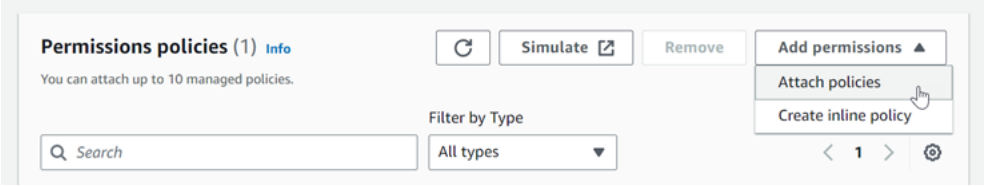
Um Ihre Funktion so zu konfigurieren, dass sie aus DynamoDB Streams liest, fügen Sie die AWSLambdaDynamoDBExecutionRole AWS verwaltete Richtlinie Ihrer Ausführungsrolle hinzu und erstellen Sie dann einen DynamoDB-Trigger.

So fügen Sie Berechtigungen hinzu und erstellen einen Auslöser

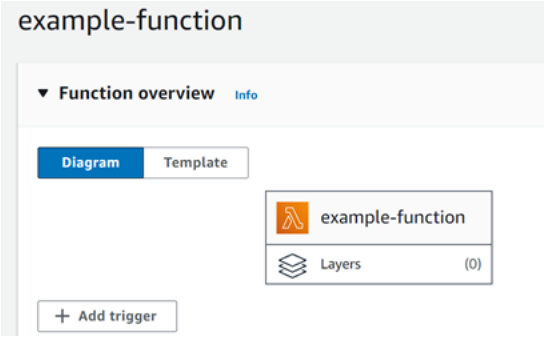
1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



5. Wählen Sie Berechtigungen hinzufügen aus und wählen Sie dann Richtlinien direkt anhängen aus.



6. Geben Sie im Suchfeld `AWSLambdaDynamoDBExecutionRole` ein. Fügen Sie diese Richtlinie zu Ihrer Ausführungsrolle hinzu. Dies ist eine AWS verwaltete Richtlinie, die die Berechtigungen enthält, die Ihre Funktion zum Lesen aus dem DynamoDB-Stream benötigt. Weitere Informationen zu dieser Richtlinie finden Sie AWSLambdaDynamoDBExecutionRolein der Referenz zu AWS verwalteten Richtlinien.
7. Gehen Sie zurück zu Ihrer Funktion in der Lambda-Konsole. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).



8. Wählen Sie einen Auslösertyp aus.
9. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Optionen für DynamoDB-Ereignisquellen:

Optionen für die Ereignisquelle

- DynamoDB-Tabelle – Die DynamoDB-Tabelle, aus der Datensätze gelesen werden sollen.
- Batchgröße – Die Anzahl der Datensätze, die in jedem Batch bis zu 10.000 an die Funktion gesendet werden sollen. Lambda übergibt alle Datensätze im Batch in einem einzigen Aufruf an die Funktion, solange die Gesamtgröße der Ereignisse nicht das Nutzlast-Limit für synchrone Aufrufe überschreitet (6 MB).
- Batchfenster – Geben Sie die maximale Zeitspanne zur Erfassung von Datensätzen vor dem Aufruf der Funktion in Sekunden an.
- Startposition – Verarbeiten Sie nur neue Datensätze oder alle vorhandenen Datensätze.
  - Neueste – Verarbeiten Sie Datensätze, die neu zum Stream hinzugefügt wurden.
  - Horizont trimmen – Verarbeiten Sie alle Datensätze im Stream.

  Nach der Verarbeitung aller vorhandenen Datensätze hat die Funktion aufgeholt und setzt die Verarbeitung neuer Datensätze fort.
- On-failure Ziel — Eine Standard-SQS-Warteschlange oder ein Standard-SNS-Thema für Datensätze, die nicht verarbeitet werden können. Wenn Lambda einen Datensatz-Batch verwirft, weil er zu alt ist oder alle Wiederholungen erschöpft hat, sendet es Details zum Batch an die Warteschlange oder das Thema.
- Wiederholungsversuche – Die maximale Anzahl von Wiederholungen von Lambda, wenn die Funktion einen Fehler zurückgibt. Dies gilt nicht für Servicefehler oder Drosselungen, bei denen der Batch die Funktion nicht erreicht hat.
- Höchstalter des Datensatzes – Das maximale Alter eines Datensatzes, den Lambda an Ihre Funktion sendet.
- Batch bei Fehler aufteilen – Wenn die Funktion einen Fehler zurückgibt, teilen Sie den Batch vor dem erneuten Versuch in zwei Teile. Ihre ursprüngliche Einstellung für die Batch-Größe bleibt unverändert.
- Gleichzeitige Batches pro Shard – Verarbeitet gleichzeitig mehrere Batches aus demselben Shard.
- Aktiviert – Auf „true“ festlegen, um die Ereignisquellenzuordnung zu aktivieren. Auf "false" festlegen, um die Verarbeitung von Datensätzen zu beenden. Lambda merkt sich den zuletzt verarbeiteten Datensatz und setzt die Verarbeitung nach erneuter Aktivierung der Zuordnung an dieser Stelle fort.

> **Note**
>
> Für GetRecords API-Aufrufe, die von Lambda als Teil von DynamoDB-Triggern aufgerufen werden, fallen keine Gebühren an.

Um die Konfiguration der Ereignisquelle zu einem späteren Zeitpunkt zu verwalten, wählen Sie den Auslöser im Designer aus.

Erstellen einer kontoübergreifenden Zuordnung von Ereignisquellen

Amazon DynamoDB unterstützt jetzt ressourcenbasierte Richtlinien. Mit dieser Funktion können Sie Daten aus einem DynamoDB-Stream in einem AWS-Konto mit einer Lambda-Ereignisfunktion in einem anderen Konto verarbeiten.

Um eine Ereignisquellenzuordnung für Ihre Lambda-Ereignisfunktion mithilfe eines DynamoDB-Streams in einem anderen zu erstellen AWS-Konto, müssen Sie den Stream mithilfe einer ressourcenbasierten Richtlinie konfigurieren, um Ihrer Lambda-Ereignisfunktion die Berechtigung zum Lesen von Datensätzen zu erteilen. Informationen dazu, wie Sie Ihren Stream für den kontoübergreifenden Zugriff konfigurieren, finden Sie unter Zugriff mit kontoübergreifenden Lambda-Funktionen teilen im Amazon DynamoDB DynamoDB-Entwicklerhandbuch.

Nachdem Sie Ihren Stream mit einer ressourcenbasierten Richtlinie konfiguriert haben, die Ihrer Lambda-Ereignisfunktion die erforderlichen Berechtigungen erteilt, erstellen Sie die Ereignisquellenzuordnung mit Ihrem kontoübergreifenden Stream-ARN. Sie finden den Stream-ARN auf der Registerkarte Exporte und Streams der Tabelle in der kontoübergreifenden DynamoDB-Konsole.

Wenn Sie die Lambda-Konsole verwenden, fügen Sie den Stream-ARN direkt in das Eingabefeld der DynamoDB-Tabelle auf der Seite zur Erstellung der Ereignisquellenzuordnung ein.

Hinweis: Cross-region Trigger werden nicht unterstützt.

Konfigurieren einer teilweisen Batch-Antwort mit DynamoDB und Lambda

Bevor Sie partielle Batch-Antworten konfigurieren, müssen Sie zunächst eine DynamoDB-Ereignisquellenzuordnung erstellen. Detaillierte Anweisungen finden Sie unter the section called “Zuordnung erstellen”.

Beim Konsumieren und Verarbeiten von Streaming-Daten aus einer Ereignisquelle werden standardmäßig Lambda-Checkpoints auf die höchste Sequenznummer eines Batches nur dann

überprüft, wenn der Batch ein voller Erfolg ist. Lambda behandelt alle anderen Ergebnisse als einen vollständigen Fehler und versucht, den Batch bis zum Wiederholungslimit zu verarbeiten. Um beim Verarbeiten von Stapeln aus einem Stream Teilerfolge zu ermöglichen, aktivieren Sie `ReportBatchItemFailures`. Das Zulassen teilweiser Erfolge kann dazu beitragen, die Anzahl der Wiederholungen in einem Datensatz zu reduzieren, verhindert jedoch nicht vollständig die Möglichkeit von Wiederholungen in einem erfolgreichen Datensatz.

Zum Aktivieren von `ReportBatchItemFailures` fügen Sie den Enum-Wert **`ReportBatchItemFailures`** der FunctionResponseTypes-Liste hinzu. Diese Liste gibt an, welche Antworttypen für Ihre Event-Funktion aktiviert sind. Sie können diese Liste konfigurieren, wenn Sie eine Zuordnung von Ereignisquellen erstellen oder aktualisieren.

> **Note**
>
> Selbst wenn Ihr Funktionscode partielle Batch-Fehlerantworten zurückgibt, werden diese Antworten nicht von Lambda verarbeitet, es sei denn, die `ReportBatchItemFailures` Funktion ist explizit für Ihre Ereignisquellenzuordnung aktiviert.

Berichtsyntax

Beim Konfigurieren von Berichten zu Batch-Elementfehlern wird die `StreamsEventResponse`-Klasse mit einer Liste von Batch-Elementfehlern zurückgegeben. Sie können ein `StreamsEventResponse`-Objekt verwenden, um die Sequenznummer des ersten fehlgeschlagenen Datensatzes im Batch zurückzugeben. Sie können auch Ihre eigene benutzerdefinierte Klasse mit der richtigen Antwortsyntax erstellen. Die folgende JSON-Struktur zeigt die erforderliche Antwortsyntax:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> Wenn das `batchItemFailures`-Array mehrere Elemente enthält, verwendet Lambda den Datensatz mit der niedrigsten Sequenznummer als Kontrollpunkt. Lambda wiederholt dann alle Datensätze ab diesem Kontrollpunkt.

Erfolgs- und Misserfolgsbedingungen

Lambda behandelt einen Batch als vollständigen Erfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere `batchItemFailure`-Liste
- Eine ungültige `batchItemFailure`-Liste
- Ein leeres `EventResponse`
- Ein ungültiges `EventResponse`

Lambda behandelt einen Batch als vollständigen Misserfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere Zeichenfolge `itemIdentifier`
- Ein ungültiges `itemIdentifier`
- Ein `itemIdentifier` mit einem falschen Schlüsselnamen

Lambda wiederholt Fehler basierend auf Ihrer Wiederholungsstrategie.

Einen Batch halbieren

Wenn Ihr Aufruf fehlschlägt und `BisectBatchOnFunctionError` eingeschaltet ist, wird der Stapel unabhängig von Ihrer `ReportBatchItemFailures`-Einstellung halbiert.

Wenn eine partielle Batch-Erfolgsantwort empfangen wird und sowohl `BisectBatchOnFunctionError` als auch `ReportBatchItemFailures` aktiviert sind, wird der Batch mit der zurückgegebenen Sequenznummer halbiert und Lambda versucht nur die verbleibenden Datensätze erneut.

Um die Implementierung der partiellen Batch-Antwortlogik zu vereinfachen, sollten Sie in Erwägung ziehen, das Batch Processor Utility von Powertools for zu verwenden AWS Lambda, das diese Komplexität automatisch für Sie erledigt.

Hier sind einige Beispiele für Funktionscodes, die die Liste der fehlgeschlagenen Nachrichten-IDs im Batch zurückgeben:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
```

```
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von. JavaScript

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }
```

```
  return { batchItemFailures: [] };
};
```

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von. TypeScript

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
```

```
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
```

```
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
```

```
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Verwenden von Powertools für AWS Lambda Batch-Prozessor

Das Batchverarbeitungsprogramm von Powertools for verarbeitet AWS Lambda automatisch die Logik für partielle Batch-Antworten und reduziert so die Komplexität der Implementierung von Batch-Fehlerberichten. Hier sind Beispiele für die Verwendung des Batch-Prozessors:

Python

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import DynamoDBStreamEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.DynamoDBStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { DynamoDBStreamEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.DynamoDBStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: DynamoDBStreamEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class DynamoDBStreamBatchHandler implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<DynamodbEvent, StreamsEventResponse> handler;

    public DynamoDBStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withDynamoDbBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
    }

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent ddbEvent, Context
 context) {
        return handler.processBatch(ddbEvent, context);
    }

    private void processMessage(DynamodbEvent.DynamodbStreamRecord
 dynamodbStreamRecord, Context context) {
        // Process the change record
    }
}
```

.NET

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
```

```
using Amazon.Lambda.DynamoDBEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;

[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class Customer
{
    public string? CustomerId { get; set; }
    public string? Name { get; set; }
    public string? Email { get; set; }
    public DateTime CreatedAt { get; set; }
}

internal class TypedDynamoDbRecordHandler : ITypedRecordHandler<Customer>
{
    public async Task<RecordHandlerResult> HandleAsync(Customer customer,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(customer.Email))
        {
            throw new ArgumentException("Customer email is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedDynamoDbRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(DynamoDBEvent _)
    {
        return TypedDynamoDbStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

Verworfene Datensätze für eine DynamoDB-Ereignisquelle in Lambda aufbewahren

Die Fehlerbehandlung für DynamoDB-Ereignisquellenzuordnungen hängt davon ab, ob der Fehler vor dem Aufrufen der Ereignisfunktion oder während des Funktionsaufrufs auftritt:

- Vor dem Aufruf: Wenn eine Lambda-Ereignisquellenzuordnung die Funktion aufgrund von Drosselung oder anderen Problemen nicht aufrufen kann, versucht sie es erneut, bis die Datensätze ablaufen oder das in der Ereignisquellenzuordnung () konfigurierte Höchstalter überschreiten. MaximumRecordAgeInSeconds
- Während des Aufrufs: Wenn die Funktion aufgerufen wird, aber einen Fehler zurückgibt, versucht Lambda es erneut, bis die Datensätze ablaufen, das Höchstalter (MaximumRecordAgeInSeconds) überschreiten oder die konfigurierte Wiederholungsquote () erreicht haben. MaximumRetryAttempts Bei Funktionsfehlern können Sie auch konfigurieren BisectBatchOnFunctionError, dass ein fehlgeschlagener Batch in zwei kleinere Batches aufgeteilt wird, wodurch fehlerhafte Datensätze isoliert und Timeouts vermieden werden. Durch das Aufteilen von Batches wird die Quote für Wiederholungen nicht verbraucht.

Wenn die Fehlerbehandlungsmaßnahmen fehlschlagen, verwirft Lambda die Datensätze und setzt die Verarbeitung von Batches aus dem Stream fort. Bei den Standardeinstellungen bedeutet dies, dass ein fehlerhafter Datensatz die Verarbeitung auf dem betroffenen Shard für bis zu einen Tag blockieren kann. Um dies zu vermeiden, konfigurieren Sie die Ereignisquellenzuordnung Ihrer Funktion mit einer angemessenen Anzahl von Wiederholungen und einem maximalen Datensatzalter, das zu Ihrem Anwendungsfall passt.

Konfigurieren von Zielen für fehlgeschlagene Aufrufe

Um Aufzeichnungen über fehlgeschlagene Aufrufe der Ereignisquellenzuordnung beizubehalten, fügen Sie der Ereignisquellenzuordnung Ihrer Ereignisfunktion ein Ziel hinzu. Jeder an das Ziel gesendete Datensatz ist ein JSON-Dokument mit Metadaten über den fehlgeschlagenen Aufruf. Bei Amazon S3-Zielen sendet Lambda auch den gesamten Aufrufdatensatz zusammen mit den Metadaten. Sie können jedes Amazon SNS SNS-Thema, jede Amazon SQS SQS-Warteschlange, jeden Amazon S3 S3-Bucket oder Kafka als Ziel konfigurieren.

Bei Amazon-S3-Zielen können Sie das Feature Amazon S3 Event Notifications verwenden, um Benachrichtigungen zu erhalten, wenn Objekte in Ihren S3-Ziel-Bucket hochgeladen werden. Sie können S3-Ereignisbenachrichtigungen auch so konfigurieren, dass eine weitere Lambda-Ereignisfunktion aufgerufen wird, um die automatische Verarbeitung fehlgeschlagener Batches durchzuführen.

Ihre Ausführungsrolle muss über Berechtigungen für das Ziel verfügen:

- Für ein SQS-Ziel: sqs: SendMessage
- Für ein SNS-Ziel: sns:Publish
- Für ein S3-Ziel: s3: und s3: PutObject ListBucket
- Für ein Kafka-Ziel: kafka-cluster: WriteData

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das angegebene Kafka-Thema. Weitere Informationen finden Sie unter the section called “Kafka-Ziel bei einem Ausfall”.

Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für ein S3-Ziel aktiviert haben, muss die Ausführungsrolle Ihrer Funktion auch die Berechtigung haben, kms: aufzurufen. GenerateDataKey Wenn sich der KMS-Schlüssel und das S3-Bucket-Ziel in einem anderen Konto befinden als Ihre Lambda-Ereignisfunktion und Ausführungsrolle, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, um kms: GenerateDataKey zuzulassen.

Gehen Sie folgendermaßen vor, um ein Ausfallziel mit der Konsole zu konfigurieren:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie als Quelle die Option Aufruf der Zuordnung von Ereignisquellen aus.
5. Wählen Sie für die Zuordnung von Ereignisquellen eine Ereignisquelle aus, die für diese Funktion konfiguriert ist.
6. Wählen Sie für Bedingung die Option Bei Ausfall aus. Für Aufrufe zur Zuordnung von Ereignisquellen ist dies die einzig akzeptierte Bedingung.
7. Wählen Sie unter Zieltyp den Zieltyp aus, an den Lambda Aufrufdatensätze sendet.
8. Wählen Sie unter Destination (Ziel) eine Ressource aus.
9. Wählen Sie Speichern.

Sie können mit AWS Command Line Interface (AWS CLI) auch ein Ziel für den Fall eines Fehlers konfigurieren. Mit dem folgenden Befehl create-event-source-mapping wird beispielsweise eine Zuordnung von Ereignisquellen mit einem SQS-Ziel zu `MyFunction` hinzugefügt:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

Der folgende update-event-source-mapping-Befehl aktualisiert eine Zuordnung von Ereignisquellen, um fehlgeschlagene Aufrufdatensätze nach zwei Wiederholungsversuchen an ein SNS-Ziel zu senden oder wenn die Datensätze älter als eine Stunde sind.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

Aktualisierte Einstellungen werden asynchron angewendet und werden erst nach Abschluss des Vorgangs in der Ausgabe berücksichtigt. Verwenden Sie den get-event-source-mapping-Befehl, um den aktuellen Status anzuzeigen.

Um ein Ziel zu entfernen, geben Sie eine leere Zeichenfolge als Argument für den `destination-config`-Parameter an:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Bewährte Methoden für die Sicherheit in Amazon S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze aus Ihrer Event-Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Lambda-Ereignisfunktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:
   a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.
   b. Wählen Sie im Richtlinien-Editor JSON aus.
   c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.
   d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.
   e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle (CLI) einer Event-Funktion eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Beispiel für einen Amazon SNS- und Amazon SQS-Aufrufsatz

Das folgende Beispiel zeigt einen Aufrufsatz, den Lambda an ein SQS- oder SNS-Ziel für einen DynamoDB-Stream sendet.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

Sie können diese Informationen verwenden, um die betroffenen Datensätze aus dem Stream für die Fehlersuche abzurufen. Die tatsächlichen Datensätze sind nicht enthalten, daher müssen Sie diesen Datensatz verarbeiten und aus dem Stream abrufen, bevor sie ablaufen und verloren gehen.

Beispiel eines Amazon S3-Aufrufdatensatzes

Das folgende Beispiel zeigt einen Aufrufsatz, den Lambda für einen DynamoDB-Stream an einen S3-Bucket sendet. Zusätzlich zu allen Feldern aus dem vorherigen Beispiel für SQS- und SNS-Ziele enthält das Feld `payload` den ursprünglichen Aufrufdatensatz als maskierte JSON-Zeichenfolge.

```
{
```

```
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

Das S3-Objekt, das den Aufrufdatensatz enthält, verwendet die folgende Namenskonvention:

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implementierung der statusbehafteten DynamoDB-Stream-Verarbeitung in Lambda

Lambda Event Functions können kontinuierliche Stream-Verarbeitungsanwendungen ausführen. Ein Stream entspricht einer unbegrenzten Menge von Daten, die kontinuierlich durch Ihre Anwendung fließen. Um Informationen aus dieser sich ständig aktualisierenden Eingabe zu analysieren, können Sie die enthaltenen Datensätze mithilfe eines zeitlich definierten Fensters binden.

Rollierende Fenster sind unterschiedliche Zeitfenster, die sich in regelmäßigen Abständen öffnen und schließen. Standardmäßig sind Lambda-Aufrufe zustandslos – Sie können sie nicht für die Verarbeitung von Daten über mehrere kontinuierliche Aufrufe hinweg ohne eine externe Datenbank verwenden. Mit rollierenden Fenstern können Sie jedoch Ihren Status über Aufrufe hinweg beibehalten.

Dieser Zustand enthält das Gesamtergebnis der Nachrichten, die zuvor für das aktuelle Fenster verarbeitet wurden. Ihr Status kann für jeden Shard maximal 1 MB betragen. Wenn er diese Größe überschreitet, wird Lambda das Fenster vorzeitig beenden.

Jeder Datensatz in einem Stream gehört zu einem bestimmten Fenster. Lambda verarbeitet jeden Datensatz mindestens einmal, garantiert jedoch nicht, dass jeder Datensatz nur einmal verarbeitet wird. In seltenen Fällen, etwa bei der Fehlerbehandlung, werden einige Datensätze möglicherweise mehrmals verarbeitet. Datensätze werden beim ersten Mal immer in der richtigen Reihenfolge verarbeitet. Wenn Datensätze mehr als einmal verarbeitet werden, werden sie nicht in der richtigen Reihenfolge verarbeitet.

Aggregation und Verarbeitung

Ihre benutzerverwaltete Funktion wird sowohl zur Aggregation als auch zur Verarbeitung der Endergebnisse dieser Aggregation aufgerufen. Lambda aggregiert alle im Fenster empfangenen Datensätze. Sie können diese Datensätze in mehreren Stapeln erhalten, jeweils als ein separater Aufruf. Jeder Aufruf erhält einen Zustand.

Wenn Sie Tumbling-Fenster verwenden, muss Ihre Antwort auf die Lambda-Ereignisfunktion daher eine `state` Eigenschaft enthalten. Wenn die Antwort keine `state`-Eigenschaft enthält, betrachtet Lambda dies als fehlgeschlagenen Aufruf. Um diese Bedingung zu erfüllen, kann Ihre Funktion ein `TimeWindowEventResponse`-Objekt zurückgeben, das die folgende JSON-Form aufweist:

Example **`TimeWindowEventResponse`**-Werte

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Für Java-Funktionen empfehlen wir, eine `Map<String, String>` zu verwenden, um den Status darzustellen.

Am Ende des Fensters `isFinalInvokeForWindow` wird das Flag so gesetzt, dass es `true` sich um den endgültigen Status handelt und dass er zur Verarbeitung bereit ist. Nach der Verarbeitung werden das Fenster und Ihr endgültiger Aufruf wird abgeschlossen, und dann wird der Zustand gelöscht.

Am Ende Ihres Fensters verwendet Lambda die endgültige Verarbeitung für Aktionen an den Aggregationsergebnissen. Ihre endgültige Verarbeitung wird synchron aufgerufen. Nach erfolgreichem Aufruf zeigt Ihre Funktion auf die Sequenznummer und die Stream-Verarbeitung wird fortgesetzt. Wenn der Aufruf nicht erfolgreich ist, unterbricht Ihre Lambda-Ereignisfunktion die weitere Verarbeitung bis zu einem erfolgreichen Aufruf.

Example DynamodbTimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
```

```
    {
       "eventID":"2",
       "eventName":"MODIFY",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "NewImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"222",
          "SizeBytes":59,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"3",
       "eventName":"REMOVE",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
```

```
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Konfiguration

Sie können rollierende Fenster konfigurieren, wenn Sie eine Ereignisquellenzuordnung erstellen oder aktualisieren. Um ein Taumelfenster zu konfigurieren, geben Sie das Fenster in Sekunden () an. TumblingWindowInSeconds Mit dem folgenden Beispielbefehl AWS Command Line Interface (AWS CLI) wird eine Quellenzuordnung für Streaming-Ereignisse erstellt, die ein Wechselfenster von 120 Sekunden hat. Die für die Aggregation und Verarbeitung definierte Lambda-Ereignisfunktion trägt den Namen. `tumbling-window-example-function`

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
```

```
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda bestimmt die rollierenden Fenstergrenzen basierend auf dem Zeitpunkt, zu dem Datensätze in den Stream eingefügt wurden. Für alle Datensätze steht ein ungefährer Zeitstempel zur Verfügung, den Lambda in Grenzbestimmungen verwendet.

Rollierende Fensteraggregationen unterstützen kein Resharding. Wenn der Shard endet, betrachtet Lambda das Fenster als geschlossen und die untergeordneten Shards beginnen ihr eigenes Fenster in einem neuen Zustand.

Rollierende Fenster unterstützen vollständig die bestehenden Wiederholungsrichtlinien `maxRetryAttempts` und `maxRecordAge`.

Example Handler.py — Aggregation und Verarbeitung

Die folgende Python-Funktion veranschaulicht, wie Sie Ihren Endzustand aggregieren und dann verarbeiten:

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Lambdaparameter für Amazon DynamoDB-SQS-Zuordnungen von Ereignisquellen

Alle Lambda-Ereignisquelltypen verwenden dieselben CreateEventSourceMappingUpdateEventSourceMappingAPI-Operationen. Allerdings gelten nur einige der Parameter für DynamoDB Streams.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
BisectBatchOnFunctionError	N	false	Keine
DestinationConfig	N	N/A	Ein Ziel der Amazon-SQS-Standardwarteschlange oder des Amazon-SNS-Standardthemas für verworfene Datensätze
Aktiviert	N	true	Keine
EventSourceArn	Y	N/A	Der ARN des Datenstroms oder eines Stream-Konsumenten
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet
FunctionName	Y	N/A	Keine
FunctionResponseTypes	N	N/A	Damit Ihre Funktion bestimmte Fehler in einem Batch meldet, beziehen Sie den

Parameter	Erforderlich	Standard	Hinweise
			Wert `ReportBatchItemFailures` in `FunctionResponseTypes` ein. Weitere Informationen finden Sie unter Konfigurieren einer teilweisen Batch-Antwort mit DynamoDB und Lambda.
MaximumBatchingWindowInSeconds	N	0	Keine
MaximumRecordAgeInSeconds	N	-1	-1 bedeutet unendlich: Fehlgeschlagene Datensätze werden wiederholt, bis der Datensatz abläuft. Das Datenaufbewahrungslimit für DynamoDB Streams beträgt 24 Stunden. Minimum: -1 Höchstwert: 604 800
MaximumRetryAttempts	N	-1	-1 bedeutet unendlich: Fehlgeschlagene Datensätze werden wiederholt, bis der Datensatz abläuft Minimum: 0 Höchstwert: 10 000.

Parameter	Erforderlich	Standard	Hinweise
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y	N/A	TRIM_HORIZON oder LATEST
TumblingWindowInSeconds	N	N/A	Minimum: 0 Maximum: 900

Verwendung der Ereignisfilterung mit einer DynamoDB-Ereignisquelle

Mithilfe der Ereignisfilterung können Sie steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Event-Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für DynamoDB-Ereignisquellen.

> **Note**
>
> DynamoDB-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `dynamodb`-Schlüssel.

Themen

- DynamoDB-Ereignis
- Filterung mit Tabellenattributen
- Filterung mit booleschen Ausdrücken
- Verwenden des Exists-Operators
- JSON-Format für DynamoDB-Filterung

DynamoDB-Ereignis

Angenommen, Sie haben eine DynamoDB-Tabelle mit dem Primärschlüssel `CustomerName` und den Attributen `AccountManager` und `PaymentTerms`. Im Folgenden wird ein Beispieldatensatz aus dem Stream Ihrer DynamoDB-Tabelle gezeigt.

```
{
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "NewImage": {
              "AccountManager": {
                  "S": "Pat Candella"
              },
              "PaymentTerms": {
                  "S": "60 days"
              },
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "SequenceNumber": "111",
          "SizeBytes": 26,
          "StreamViewType": "NEW_IMAGE"
      }
  }
```

Um anhand der Schlüssel- und Attributwerte in Ihrer DynamoDB-Tabelle zu filtern, verwenden Sie den dynamodb-Schlüssel im Datensatz. Die folgenden Abschnitte enthalten Beispiele für verschiedene Filtertypen.

Filtern mit Tabellenschlüsseln

Angenommen, Sie möchten, dass Ihre Event-Funktion nur die Datensätze verarbeitet, deren Primärschlüssel "AnyCompany Industries" `CustomerName` ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
     "Filters": [
          {
              "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
```

```
          }
      ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
     "dynamodb": {
          "Keys": {
              "CustomerName": {
                  "S": [ "AnyCompany Industries" ]
                  }
              }
          }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer AWS SAM Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
```

```
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

Filterung mit Tabellenattributen

Mit DynamoDB können Sie auch die Schlüssel `NewImage` und `OldImage` verwenden, um nach Attributwerten zu filtern. Angenommen, Sie möchten Datensätze filtern, bei denen das `AccountManager` Attribut im neuesten Tabellenbild „Pat Candella“ oder „Shirley Rodriguez“ lautet. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer Vorlage hinzufügen. AWS SAM

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

Filterung mit booleschen Ausdrücken

Sie können Filter auch mithilfe von booleschen UND-Ausdrücken erstellen. Diese Ausdrücke können sowohl die Schlüssel- als auch die Attributparameter Ihrer Tabelle enthalten. Angenommen, Sie möchten Datensätze filtern, bei denen der `NewImage`-Wert von `AccountManager` „Pat Candella“ und der `OldImage`-Wert „Terry Whitlock“ ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer Vorlage hinzufügen. AWS SAM

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
```

```
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
[ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> Die DynamoDB-Ereignisfilterung unterstützt nicht die Verwendung numerischer Operatoren (numerisches Gleichheitszeichen und numerischer Bereich). Auch wenn Elemente in Ihrer Tabelle als Zahlen gespeichert sind, werden diese Parameter im JSON-Datensatzobjekt in Strings umgewandelt.

Verwenden des Exists-Operators

Aufgrund der Struktur von JSON-Ereignisobjekten aus DynamoDB ist bei der Verwendung des Operators „Exists“ besondere Vorsicht geboten. Der Exists-Operator funktioniert nur bei Blattknoten im Ereignis-JSON. Wenn Ihr Filtermuster also Exists verwendet, um nach einem Zwischenknoten zu suchen, wird es nicht funktionieren. Betrachten Sie das folgende DynamoDB-Tabellenelement:

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

Möglicherweise möchten Sie ein Filtermuster wie das folgende erstellen, das nach Ereignissen sucht, die `"Organizations"` enthalten:

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

Dieses Filtermuster würde jedoch nie eine Übereinstimmung ergeben, da `"Organizations"` kein Blattknoten ist. Das folgende Beispiel zeigt, wie der Operator „Exists“ richtig verwendet wird, um das gewünschte Filtermuster zu erstellen:

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

JSON-Format für DynamoDB-Filterung

Um Ereignisse aus DynamoDB-Quellen richtig zu filtern, müssen sowohl das Datenfeld als auch Ihre Filterkriterien für das Datenfeld (`dynamodb`) im gültigen JSON-Format vorliegen. Wenn eines der Felder kein gültiges JSON-Format hat, verwirft Lambda die Nachricht oder gibt eine Ausnahme aus. In der folgenden Tabelle ist das Verhalten zusammengefasst:

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Non-JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuordnung eine Ausnahme aus. Das Filtermuster für Dateneigenschaften muss ein gültiges JSON-Format haben.
Non-JSON	Gültiges JSON	Lambda verwirft den Datensatz.
Non-JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
Non-JSON	Non-JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuordnung eine Ausnahme aus. Das Filtermuster für Dateneigenschaften muss ein gültiges JSON-Format haben.

Tutorial: Verwenden AWS Lambda mit Amazon DynamoDB DynamoDB-Streams

In diesem Tutorial erstellen Sie eine Lambda-Ereignisfunktion, um Ereignisse aus einem Amazon DynamoDB DynamoDB-Stream zu verarbeiten.

Voraussetzungen

Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle

Erstellen Sie die Ausführungsrolle, die Ihrer Funktion die Berechtigung zum Zugriff auf AWS Ressourcen erteilt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Vertrauenswürdige Entität – Lambda.
   - Berechtigungen — AWSLambdaDynamoDBExecutionRole.
   - Role name (Name der Rolle – **`lambda-dynamodb-role`**.

Der AWSLambdaDynamoDBExecutionRolehat die Berechtigungen, die die Funktion benötigt, um Elemente aus DynamoDB zu lesen und Protokolle in Logs zu CloudWatch schreiben.

Erstellen Sie die Event-Funktion

Erstellen Sie eine Lambda-Ereignisfunktion, die Ihre DynamoDB-Ereignisse verarbeitet. Der Funktionscode schreibt einige der eingehenden Ereignisdaten in Logs. CloudWatch

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines DynamoDB-Ereignisses mit Lambda unter Verwendung. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Konsumieren eines DynamoDB-Ereignisses mit Lambda unter Verwendung. TypeScript

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data
```

```
            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Python.

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())
```

```
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

Um die Event-Funktion zu erstellen

1. Kopieren Sie den Beispiel-Code in eine Datei mit dem Namen `example.js`.
2. Erstellen Sie ein Bereitstellungspaket.

```
zip function.zip example.js
```

3. Erstellen Sie mit dem `create-function` Befehl eine Lambda-Ereignisfunktion.

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs24.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

Testen Sie die Lambda-Ereignisfunktion

In diesem Schritt rufen Sie Ihre Lambda-Ereignisfunktion manuell mit dem `invoke` AWS Lambda CLI-Befehl und dem folgenden DynamoDB-Beispielereignis auf. Kopieren Sie den folgenden Code in eine Datei mit dem Namen `input.txt`.

Example input.txt

```
{
```

```
    "Records":[
       {
          "eventID":"1",
          "eventName":"INSERT",
          "eventVersion":"1.0",
          "eventSource":"aws:dynamodb",
          "awsRegion":"us-east-1",
          "dynamodb":{
             "Keys":{
                "Id":{
                   "N":"101"
                }
             },
             "NewImage":{
                "Message":{
                   "S":"New item!"
                },
                "Id":{
                   "N":"101"
                }
             },
             "SequenceNumber":"111",
             "SizeBytes":26,
             "StreamViewType":"NEW_AND_OLD_IMAGES"
          },
          "eventSourceARN":"stream-ARN"
       },
       {
          "eventID":"2",
          "eventName":"MODIFY",
          "eventVersion":"1.0",
          "eventSource":"aws:dynamodb",
          "awsRegion":"us-east-1",
          "dynamodb":{
             "Keys":{
                "Id":{
                   "N":"101"
                }
             },
             "NewImage":{
                "Message":{
                   "S":"This item has changed"
                },
                "Id":{
```

```
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"222",
          "SizeBytes":59,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"3",
       "eventName":"REMOVE",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"333",
          "SizeBytes":38,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    }
 ]
```

```
}
```

Führen Sie den Befehl `invoke` aus.

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Funktion gibt die Zeichenfolge `message` im Antworttext zurück.

Überprüfen Sie die Ausgabe in der Datei `outputfile.txt`.

Erstellen einer DynamoDB-Tabelle mit einem aktivierten Stream

Erstellen einer Amazon-DynamoDB-Tabelle mit einem aktivierten Stream.

So erstellen Sie eine DynamoDB-Tabelle

1. Öffnen Sie die DynamoDB-Konsole.
2. Wählen Sie Create table aus.
3. Erstellen Sie eine Tabelle mit den folgenden Einstellungen:
   - Tabellenname – **`lambda-dynamodb-stream`**
   - Primärschlüssel – **`id`** (Zeichenfolge)
4. Wählen Sie Erstellen aus.

So aktivieren Sie Streams

1. Öffnen Sie die DynamoDB-Konsole.
2. Wählen Sie Tables (Tabellen) aus.
3. Wählen Sie die Tabelle lambda-dynamodb-stream aus.
4. Wählen Sie unter Exports and streams (Exporte und Streams) die Option DynamoDB stream details (Details zu DynamoDB-Streams) aus.

5. Wählen Sie Turn on (Einschalten) aus.
6. Wählen Sie als Ansichtstyp die Option Nur Schlüsselattribute aus.
7. Wählen Sie Stream einschalten.

Notieren Sie sich den Stream-ARN. Sie benötigen dies im nächsten Schritt, wenn Sie den Stream mit Ihrer Lambda-Ereignisfunktion verknüpfen. Weitere Informationen zur Aktivierung von Streams finden Sie unter Erfassen von Tabellenaktivitäten mit DynamoDB Streams.

Fügen Sie eine Ereignisquelle hinzu in AWS Lambda

Erstellen Sie eine Ereignisquellenzuordnung in AWS Lambda. Diese Ereignisquellenzuordnung verknüpft den DynamoDB-Stream mit Ihrer Lambda-Ereignisfunktion. Nachdem Sie diese Ereignisquellenzuordnung erstellt haben, AWS Lambda beginnt die Abfrage des Streams.

Führen Sie den Befehl AWS CLI `create-event-source-mapping` aus. Notieren Sie sich die UUID, nachdem Sie den Befehl ausgeführt haben. Sie benötigen diese UUID, um in Befehlen auf die Ereignisquellenzuordnung zu verweisen (beispielsweise beim Löschen der Ereignisquellen-Zuweisung).

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

Dadurch wird eine Zuordnung zwischen dem angegebenen DynamoDB-Stream und der Lambda-Ereignisfunktion erstellt. Sie können einen DynamoDB-Stream mehreren Lambda-Ereignisfunktionen zuordnen und dieselbe Lambda-Ereignisfunktion mehreren Streams zuordnen. Die Lambda-Ereignisfunktionen teilen sich jedoch den Lesedurchsatz für den Stream, den sie gemeinsam nutzen.

Sie erhalten die Liste der Ereignisquellen-Zuweisungen, indem Sie folgenden Befehl ausführen.

```
aws lambda list-event-source-mappings
```

Die Liste gibt alle von Ihnen erstellten Ereignisquellenzuordnungen zurück und für jede Zuordnung zeigt sie u. a. den `LastProcessingResult`. Dieses Feld wird verwendet, um eine informative Meldung bereitzustellen, wenn Probleme auftreten. Werte wie `No records processed` (gibt an, dass die Abfrage nicht gestartet AWS Lambda wurde oder dass der Stream keine Datensätze enthält) und `OK` (gibt an, dass Datensätze AWS Lambda erfolgreich aus dem Stream gelesen und Ihre Lambda-Ereignisfunktion aufgerufen wurden) weisen darauf hin, dass keine Probleme vorliegen. Bei Problemen erhalten Sie eine Fehlermeldung.

Bei einer großen Anzahl von Ereignisquellen-Zuweisungen verwenden Sie den Funktionsnamen-Parameter, um die Ergebnisse einzugrenzen.

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

Testen der Einrichtung

Testen Sie die durchgehende Erfahrung. Beim Aktualisieren der Tabelle schreibt DynamoDB Ereignisdatensätze in den Stream. Beim AWS Lambda Abrufen des Streams werden neue Datensätze im Stream erkannt und Ihre Lambda-Ereignisfunktion in Ihrem Namen aufgerufen, indem Ereignisse an die Funktion übergeben werden.

1. Fügen Sie in der DynamoDB-Konsole Elemente hinzu, aktualisieren und löschen Sie diese und fügen Sie diese hinzu. DynamoDB schreibt Datensätze dieser Aktionen in den Stream.
2. AWS Lambda fragt den Stream ab und wenn es Aktualisierungen am Stream erkennt, ruft es Ihre Lambda-Ereignisfunktion auf, indem es die im Stream gefundenen Ereignisdaten weitergibt.
3. Ihre Funktion wird in Amazon ausgeführt und erstellt Protokolle CloudWatch. Sie können die in der CloudWatch Amazon-Konsole gemeldeten Protokolle überprüfen.

Nächste Schritte

In diesem Tutorial haben Sie die Grundlagen der Verarbeitung von DynamoDB-Stream-Ereignissen mit Lambda kennengelernt. Bei Produktions-Workloads sollten Sie erwägen, eine Logik für teilweise Batch-Antworten zu implementieren, um einzelne Datensatzfehler effizienter zu handhaben. Das Batchverarbeitungsprogramm von Powertools for AWS Lambda ist in Python TypeScript, .NET und Java verfügbar und bietet hierfür eine robuste Lösung, die die Komplexität von partiellen Batch-Antworten automatisch handhabt und die Anzahl der Wiederholungen für erfolgreich verarbeitete Datensätze reduziert.

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre. AWS-Konto

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie die von Ihnen erstellte Tabelle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie **`delete`** in das Textfeld ein.
5. Wählen Sie Delete Table (Tabelle löschen).

Verarbeiten Sie Amazon EC2 EC2-Lebenszyklusereignisse mit einer Lambda-Ereignisfunktion

Sie können AWS Lambda es verwenden, um Lebenszyklusereignisse aus Amazon Elastic Compute Cloud zu verarbeiten und Amazon EC2 EC2-Ressourcen zu verwalten. Amazon EC2 sendet Ereignisse an Amazon EventBridge (CloudWatch Events) für Lebenszyklusereignisse, z. B. wenn sich der Status einer Instance ändert, wenn ein Amazon Elastic Block Store-Volume-Snapshot abgeschlossen ist oder wenn die Beendigung einer Spot-Instance geplant ist. Sie konfigurieren EventBridge (CloudWatch Events) so, dass diese Ereignisse zur Verarbeitung an eine Lambda-Ereignisfunktion weitergeleitet werden.

EventBridge (CloudWatch Events) ruft Ihre Lambda-Ereignisfunktion asynchron mit dem Ereignisdokument von Amazon EC2 auf.

Example Instance-Lebenszyklusereignis

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

Einzelheiten zur Konfiguration von Ereignissen finden Sie unter Rufen Sie eine Lambda-Ereignisfunktion nach einem Zeitplan auf. Eine Beispielfunktion, die Amazon EBS-Snapshot-Benachrichtigungen verarbeitet, finden Sie unter EventBridge Scheduler for Amazon EBS.

Sie können das AWS SDK auch verwenden, um Instances und andere Ressourcen mit der Amazon EC2 EC2-API zu verwalten.

Erteilen von Berechtigungen für EventBridge (CloudWatch Ereignisse)

Um Lebenszyklusereignisse von Amazon EC2 verarbeiten zu können, benötigt EventBridge (CloudWatch Events) die Erlaubnis, Ihre Funktion aufzurufen. Diese Berechtigung stammt aus der ressourcenbasierten Richtlinie der Funktion. Wenn Sie die Konsole EventBridge (CloudWatch Events) verwenden, um einen Event-Trigger zu konfigurieren, aktualisiert die Konsole die ressourcenbasierte Richtlinie in Ihrem Namen. Andernfalls fügen Sie eine Anweisung wie die folgende hinzu:

Example Ressourcenbasierte Richtlinienanweisung für Amazon-EC2-Lebenszyklusbenachrichtigungen

```
{
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

Verwenden Sie den Befehl, um eine Anweisung hinzuzufügen. `add-permission` AWS CLI

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

Wenn Ihre Funktion das AWS SDK zur Verwaltung von Amazon EC2 EC2-Ressourcen verwendet, fügen Sie der Ausführungsrolle der Funktion Amazon EC2 EC2-Berechtigungen hinzu.

Verarbeiten von Application Load Balancer-Anfragen mit Lambda

Sie können eine Lambda-Ereignisfunktion verwenden, um Anfragen von einem Application Load Balancer zu verarbeiten. Elastic Load Balancing unterstützt Lambda Event Functions als Ziel für einen Application Load Balancer. Sie können Load Balancer-Regeln zum Weiterleiten von HTTP-Anforderungen an eine Funktion verwenden, basierend auf Pfad- oder Header-Werten. Verarbeiten Sie die Anfrage und geben Sie eine HTTP-Antwort von Ihrer Lambda-Ereignisfunktion zurück.

Elastic Load Balancing ruft Ihre Lambda Event Function synchron mit einem Ereignis auf, das den Anforderungstext und die Metadaten enthält.

Example Anforderungsereignis von einem Application Load Balancer

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
    "isBase64Encoded": False
```

```
}
```

Ihre Funktion verarbeitet das Ereignis und gibt ein Antwortdokument in JSON an den Load Balancer zurück. Elastic Load Balancing konvertiert das Dokument in eine HTTP-Erfolgs- oder Fehlerantwort und gibt es an den Benutzer zurück.

Example Format des Antwortdokuments

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

Um eine Application Load Balancer als Funktionsauslöser zu konfigurieren, gewähren Sie Elastic Load Balancing die Berechtigung zur Ausführung der Funktion, erstellen eine Zielgruppe, die Anforderungen an die Funktion weiterleitet und fügen dem Load Balancer eine Regel hinzu, die Anforderungen an die Zielgruppe sendet.

Sie können den Befehl `add-permission` verwenden, um der ressourcenbasierten Richtlinie Ihrer Funktion eine Berechtigungsanweisung hinzuzufügen.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

Anweisungen zum Konfigurieren des Application Load Balancer-Listeners und der Zielgruppe finden Sie unter Lambda-Funktionen als Ziel im Benutzerhandbuch für Application Load Balancer.

Event Handler von Powertools für AWS Lambda

Der Event-Handler aus dem Powertools for AWS Lambda-Toolkit bietet Routing, Middleware, CORS-Konfiguration, Generierung von OpenAPI-Spezifikationen, Anforderungsvalidierung, Fehlerbehandlung und andere nützliche Funktionen beim Schreiben von Lambda-Ereignisfunktionen, die von einem Application Load Balancer aufgerufen werden. Das Dienstprogramm für den Ereignishandler ist für Python verfügbar. Weitere Informationen finden Sie unter Event Handler REST API in der Dokumentation zu Powertools for AWS Lambda (Python).

Python

```
import requests
from requests import Response

from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.event_handler import ALBResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing import LambdaContext

tracer = Tracer()
logger = Logger()
app = ALBResolver()


@app.get("/todos")
@tracer.capture_method
def get_todos():
    todos: Response = requests.get("https://jsonplaceholder.typicode.com/todos")
    todos.raise_for_status()

    # for brevity, we'll limit to the first 10 only
    return {"todos": todos.json()[:10]}


# You can continue to use other utilities just as before
@logger.inject_lambda_context(correlation_id_path=correlation_paths.APPLICATION_LOAD_BALANCER)
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

Rufen Sie eine Lambda-Ereignisfunktion nach einem Zeitplan auf

Amazon EventBridge Scheduler ist ein serverloser Scheduler, mit dem Sie Aufgaben von einem zentralen, verwalteten Service aus erstellen, ausführen und verwalten können. Mit EventBridge Scheduler können Sie Zeitpläne mithilfe von Cron- und Rate-Ausdrücken für wiederkehrende Muster erstellen oder einmalige Aufrufe konfigurieren. Sie können flexible Zeitfenster für die Zustellung einrichten, Wiederholungslimits definieren und die maximale Aufbewahrungszeit für unverarbeitete Ereignisse festlegen.

Wenn Sie EventBridge Scheduler mit Lambda einrichten, ruft EventBridge Scheduler Ihre Lambda-Ereignisfunktion asynchron auf. Sie können einen Zeitplan von der Lambda-Konsole (mithilfe des Trigger-Flows hinzufügen), der EventBridge Scheduler-Konsole AWS CLI, dem oder oder aus erstellen. CloudFormation Auf dieser Seite wird erklärt, wie Sie den EventBridge Scheduler verwenden, um eine Lambda-Ereignisfunktion nach einem Zeitplan aufzurufen.

Einrichten der Ausführungsrolle

Wenn Sie einen neuen Zeitplan erstellen, muss der EventBridge Scheduler über die Berechtigung verfügen, seinen API-Zielvorgang in Ihrem Namen aufzurufen. Sie gewähren EventBridge Scheduler diese Berechtigungen mithilfe einer Ausführungsrolle. Die Berechtigungsrichtlinie, die Sie der Ausführungsrolle Ihres Zeitplans hinzufügen, definiert die erforderlichen Berechtigungen. Diese Berechtigungen hängen von der Ziel-API ab, die EventBridge Scheduler aufrufen soll.

Wenn Sie mit der Lambda-Konsole oder der EventBridge Scheduler-Konsole einen Zeitplan erstellen, richtet EventBridge Scheduler automatisch eine Ausführungsrolle auf der Grundlage Ihres ausgewählten Ziels ein. Wenn Sie einen Zeitplan mit einem der EventBridge Scheduler-SDKs, dem oder, erstellen möchten, müssen Sie über eine vorhandene Ausführungsrolle verfügen AWS CLI CloudFormation, die die Berechtigungen gewährt, die EventBridge Scheduler zum Aufrufen eines Ziels benötigt. Weitere Informationen zum manuellen Einrichten einer Ausführungsrolle für Ihren Zeitplan finden Sie unter Einrichten einer Ausführungsrolle im EventBridge Scheduler-Benutzerhandbuch.

Erstellen Sie einen Zeitplan über die Lambda-Konsole

Verwenden Sie den Trigger-Add-Flow in der Lambda-Konsole, um einen EventBridge Scheduler-Zeitplan zu erstellen, ohne die Konsole zu verlassen.

So erstellen Sie einen Scheduler-Trigger mit der Lambda-Konsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Wählen Sie die Funktion aus, die Sie planen möchten.
3. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).
4. Wählen Sie in der Dropdownliste für die Triggerquelle die Option Scheduler aus.
5. Konfigurieren Sie die Zeitplaneinstellungen:
   a. Name des Zeitplans — Geben Sie einen Namen für Ihren Zeitplan ein.
   b. Zeitplangruppe — Wählen Sie eine Zeitplangruppe oder verwenden Sie sie`default`.
   c. Zeitplantyp — Wählen Sie eine der folgenden Optionen:
      - One-time Zeitplan — Ruft die Funktion einmal am angegebenen Datum und zur angegebenen Uhrzeit auf.
      - Wiederkehrender Zeitplan — Ruft die Funktion wiederholt mit einem Cron- oder Rate-Ausdruck auf.
   d. (Optional) Geben Sie eine JSON-Nutzlast ein. Wenn Sie keine Nutzlast eingeben, verwendet EventBridge Scheduler ein leeres Ereignis, um die Funktion aufzurufen.
   e. Konfigurieren Sie die Ausführungsrolle. Damit der EventBridge Scheduler eine neue Ausführungsrolle für Sie erstellt, wählen Sie Neue Rolle für diesen Zeitplan erstellen. Geben Sie dann einen Namen für Rollenname ein. Wenn Sie diese Option wählen, erstellt EventBridge Scheduler eine Richtlinie mit den erforderlichen Berechtigungen und fügt sie der Rolle hinzu.
6. Wählen Sie Hinzufügen aus.

> **Tip**
>
> Sie können auch nach,, oder suchen **`schedule`** **`crontime`**, **`recurring`** um diesen Trigger im Ablauf „Trigger hinzufügen“ zu finden.

Sie können auch die EventBridge Scheduler-Konsole verwenden, um einen Zeitplan zu erstellen. Mit der EventBridge Scheduler-Konsole können Sie zusätzliche Optionen wie flexible Zeitfenster und Warteschlangen mit unerlaubten Briefen konfigurieren.

EventBridge Scheduler-Trigger anzeigen

Alle EventBridge Scheduler-Zeitpläne, die auf Ihre Funktion abzielen, werden im Abschnitt Trigger der Funktionskonfigurationsseite unter der Bezeichnung Scheduler angezeigt. Dazu gehören Zeitpläne, die aus einer der folgenden Quellen erstellt wurden:

- Die Lambda-Konsole (Trigger-Flow hinzufügen)
- Die EventBridge Scheduler-Konsole
- Das SDK AWS CLI, CloudFormation, AWS SAM (unter Verwendung des `ScheduleV2` Ereignistyps) oder AWS CDK (unter Verwendung des `aws-scheduler` Moduls)

> **Note**
>
> Wenn ein Zeitplan einen vom Kunden verwalteten Schlüssel für die Verschlüsselung mit einem UTI-Ziel (Universal Target Invocation) verwendet, wird dieser möglicherweise nicht in der Triggerliste angezeigt. Verwenden Sie die EventBridge Scheduler-Konsole, um diese Zeitpläne anzuzeigen.

Um zu überprüfen, ob der EventBridge Scheduler die Funktion aufgerufen hat, überprüfen Sie die CloudWatch Amazon-Protokolle der Funktion.

Zugehörige Ressourcen

Weitere Informationen zu EventBridge Scheduler finden Sie im Folgenden:

- EventBridge Scheduler-Benutzerhandbuch
- EventBridge Scheduler-API-Referenz
- EventBridge Preise für Scheduler

Verwenden AWS Lambda mit AWS IoT

AWS IoT bietet sichere Kommunikation zwischen mit dem Internet verbundenen Geräten (z. B. Sensoren) und der Cloud. AWS Auf diese Weise können Sie die Telemetriedaten von mehreren Geräten erfassen, speichern und analysieren.

Sie können AWS IoT Regeln erstellen, mit denen Ihre Geräte interagieren können. AWS-Services Die AWS IoT Rules Engine bietet eine SQL-based Sprache, um Daten aus Nachrichtennutzlasten auszuwählen und die Daten an andere Dienste wie Amazon S3, Amazon DynamoDB und zu senden. AWS Lambda Sie definieren eine Regel zum Aufrufen einer Lambda-Ereignisfunktion, wenn Sie einen anderen AWS Dienst oder einen Drittanbieterdienst aufrufen möchten.

Wenn eine eingehende IoT-Nachricht die Regel auslöst, AWS IoT ruft Ihre Lambda-Ereignisfunktion asynchron auf und leitet Daten aus der IoT-Nachricht an die Funktion weiter.

Das folgende Beispiel zeigt einen Feuchtigkeitsablesung von einem Gewächshaussensor. Die Werte row und pos bestimmen die Position des Sensors. Dieses Beispielereignis basiert auf dem Gewächshaustyp in den AWS IoT -Regel-Tutorials.

Example AWS IoT Nachrichtenereignis

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

Bei asynchronem Aufruf stellt Lambda die Nachricht in die Warteschlange und wiederholt den Vorgang, wenn Ihre Funktion einen Fehler zurückgibt. Konfigurieren Sie Ihre Funktion mit einem Ziel, um Ereignisse beizubehalten, die Ihre Funktion nicht verarbeiten konnte.

Sie müssen dem AWS IoT Dienst die Erlaubnis erteilen, Ihre Lambda-Ereignisfunktion aufzurufen. Sie können den Befehl `add-permission` verwenden, um der ressourcenbasierten Richtlinie Ihrer Funktion eine Berechtigungsanweisung hinzuzufügen.

> **Important**
>
> Wenn Sie eine Berechtigung für einen AWS IoT Prinzipal hinzufügen, ohne das `source-arn` Oder anzugeben`source-account`, kann jeder AWS-Konto , der mit Ihrer AWS Lambda

Aktion eine Regel erstellt, Regeln aktivieren, von denen aus Ihre AWS Lambda Event-Funktion aufgerufen wird. AWS IoT Weitere Informationen finden Sie unter AWS Lambda Berechtigungen.

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com \
--source-arn arn:aws:iot:us-east-1:123456789012:rule/my-rule \
--source-account 123456789012
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource
\":\"arn:aws:lambda:us-east-1:123456789012:function:my-function\",\"Condition\":
{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:iot:us-east-1:123456789012:rule/my-rule\"},
\"StringEquals\":{\"AWS:SourceAccount\":\"123456789012\"}}}"
}
```

Weitere Informationen zur Verwendung von Lambda mit AWS IoT finden Sie unter AWS Lambda Regel erstellen.

Verarbeiten von Datensätzen aus Amazon Kinesis Data Streams mit Lambda

Sie können eine Lambda-Ereignisfunktion verwenden, um Datensätze in einem Amazon Kinesis Kinesis-Datenstream zu verarbeiten. Sie können eine Lambda-Ereignisfunktion einem Kinesis Data Streams Streams-Verbraucher mit gemeinsamem Durchsatz (Standard-Iterator) oder einem dedizierten Durchsatzverbraucher mit erweitertem Fan-Out zuordnen. Bei Standard-Iteratoren fragt Lambda jeden Shard in Ihrem Kinesis-Stream nach Datensätzen ab, die das HTTP-Protokoll verwenden. Die Ereignisquellenzuordnung teilt den Lesedurchsatz mit anderen Konsumenten des Shards zusammen.

Weitere Informationen zu Kinesis-Datenströmen finden Sie unter Daten aus Amazon Kinesis Data Streams.

Um zu beginnen, sehen Sie sich the section called “Zuordnung erstellen” an. Sie können Parameter für die Zuordnung von Ereignisquellen konfigurieren, Tumbling-Fenster für die Aggregation verwenden und Beispielfunktionscode anzeigen.

> **Note**
>
> Kinesis berechnet Gebühren für jeden Shard, sowie bei verbessertem Rundsenden für Daten, die aus dem Stream gelesen werden. Details zu den Preisen finden Sie unter Amazon-Kinesis- Preise.

Abfragen und Stapeln von Streams

Lambda liest Datensätze aus dem Datenstrom und ruft Ihre Funktion synchron mit einem Ereignis auf, das Stream-Datensätze enthält. Lambda liest Datensätze in Batches und ruft Ihre Funktion auf, um Datensätze aus dem Batch zu verarbeiten. Jeder Batch enthält Datensätze aus einem einzelnen shard/data Stream.

Ihre Lambda Event Function ist eine Verbraucheranwendung für Ihren Datenstrom. Sie verarbeitet jeweils einen Batch Datensätzen aus jedem Shard. Sie können eine Lambda-Ereignisfunktion einem Verbraucher mit gemeinsamem Durchsatz (Standard-Iterator) oder einem Verbraucher mit dediziertem Durchsatz und erweitertem Fan-Out zuordnen.

- Standard-Iterator: Lambda fragt jeden Shard in Ihrem Kinesis-Stream mit einer Basisrate von einmal pro Sekunde nach Datensätzen ab. Wenn mehr Datensätze verfügbar sind, verarbeitet

Lambda Batches, bis die Funktion mit dem Stream gleichzieht. Die Ereignisquellenzuordnung teilt den Lesedurchsatz mit anderen Konsumenten des Shards zusammen.

- Erweitertes Rundsenden: Um die Latenz zu minimieren und den Lesedurchsatz zu maximieren, erstellen Sie einen Daten-Stream-Konsumenten mit erweitertem Rundsenden. Stream-Konsumenten mit erweitertem Rundsenden erhalten eine dedizierte Verbindung für jeden Shard, der keine Auswirkungen auf andere Anwendungen hat, die aus dem Stream lesen. Stream-Verbraucher reduzieren HTTP/2 die Latenz, indem sie Datensätze über eine langlebige Verbindung an Lambda weiterleiten und Anforderungsheader komprimieren. Sie können mit der RegisterStreamConsumerKinesis-API einen Stream-Consumer erstellen.

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

Um die Geschwindigkeit zu erhöhen, mit der Ihre Funktion Datensätze verarbeitet, fügen Sie Ihrem Datenstrom Shards hinzu. Lambda verarbeitet Datensätze in jedem Shard in der Reihenfolge. Es beendet die Verarbeitung zusätzlicher Datensätze in einem Shard, wenn Ihre Funktion einen Fehler zurückgibt. Mehr Shards bedeutet, dass mehr Stapel verarbeitet und gleichzeitig die Auswirkungen von Fehlern auf die Nebenläufigkeit verringert werden.

Wenn Ihre Funktion nicht hochskalieren kann, um alle gleichzeitigen Stapel zu verarbeiten, fordern Sie eine Kontingenterhöhung an oder reservieren Sie Gleichzeitigkeit für Ihre Funktion.

Standardmäßig ruft Lambda Ihre Funktion auf, sobald Datensätze verfügbar sind. Wenn der Batch, den Lambda aus der Ereignisquelle liest, nur einen Datensatz enthält, sendet Lambda nur einen Datensatz an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle anweisen, Datensätze bis zu 5 Minuten lang zu

puffern, indem Sie ein Batch-Fenster konfigurieren. Bevor die Funktion aufgerufen wird, liest Lambda so lange Datensätze aus der Ereignisquelle, bis es einen vollständigen Batch erfasst hat, das Batch-Verarbeitungsfenster abläuft oder der Batch die Nutzlastgrenze von 6 MB erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Lambda wartet mit dem Senden des nächsten zu verarbeitenden Stapels nicht, bis ggf. konfigurierte Erweiterungen abgeschlossen sind. Anders ausgedrückt: Ihre Erweiterungen werden möglicherweise weiter ausgeführt, während Lambda den nächsten Stapel von Datensätzen verarbeitet. Dies kann zu Drosselungsproblemen führen, wenn Sie gegen eine Einstellung oder gegen einen Grenzwert im Zusammenhang mit der Parallelität Ihres Kontos verstoßen. Um zu erkennen, ob möglicherweise ein Problem vorliegt, müssen Sie Ihre Funktionen überwachen sowie überprüfen, ob für Ihre Zuordnung von Ereignisquellen unerwartet hohe Parallelitätsmetriken vorliegen. Aufgrund der kurzen Zeit zwischen den Aufrufen kann Lambda kurzzeitig eine höhere Gleichzeitigkeitsnutzung als die Anzahl der Shards melden. Dies kann sogar für Lambda Event Functions ohne Erweiterungen gelten.

Konfigurieren Sie die ParallelizationFactorEinstellung so, dass ein Shard eines Kinesis-Datenstroms mit mehr als einem Lambda-Aufruf gleichzeitig verarbeitet wird. Sie können die Anzahl der gleichzeitigen Batches angeben, die Lambda von einem Shard abfragt, indem Sie einen Parallelisierungsfaktor von 1 (Standard) bis 10 verwenden. Wenn Sie beispielsweise `ParallelizationFactor` auf 2 setzen, können Sie maximal 200 gleichzeitige Lambda-Aufrufe haben, um 100 Kinesis-Datensplitter zu verarbeiten (obwohl Sie in der Praxis möglicherweise unterschiedliche Werte für die Metrik sehen). `ConcurrentExecutions` Dies hilft, den Verarbeitungsdurchsatz hochzuskalieren, wenn das Datenvolumen flüchtig ist und `IteratorAge` hoch ist. Wenn Sie die Anzahl gleichzeitiger Batches pro Shard erhöhen, stellt Lambda weiterhin die Auftragsverarbeitung auf Partitionsschlüsselebene sicher.

Sie können `ParallelizationFactor` auch mit der Kinesis-Aggregation verwenden. Das Verhalten der Zuordnung von Ereignisquellen hängt davon ab, ob Sie das erweiterte Rundsenden verwenden:

- Ohne erweitertes Rundsenden: Alle Ereignisse innerhalb eines aggregierten Ereignisses müssen denselben Partitionsschlüssel haben. Der Partitionsschlüssel muss außerdem mit dem des aggregierten Ereignisses übereinstimmen. Wenn die Ereignisse innerhalb des aggregierten Ereignisses unterschiedliche Partitionsschlüssel haben, kann Lambda nicht garantieren, dass die Ereignisse in der richtigen Reihenfolge nach Partitionsschlüssel verarbeitet werden.
- Mit verbessertem Rundsenden: Zunächst dekodiert Lambda das aggregierte Ereignis in seine einzelnen Ereignisse. Das aggregierte Ereignis kann einen anderen Partitionsschlüssel haben als die darin enthaltenen Ereignisse. Ereignisse, die nicht dem Partitionsschlüssel entsprechen, werden jedoch gelöscht und gehen verloren. Lambda verarbeitet diese Ereignisse nicht und sendet sie nicht an ein konfiguriertes Fehlerziel.

Beispielereignis

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
```

```
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        }
    ]
}
```

Verarbeiten von Amazon Kinesis Data Streams-Datensätzen mit Lambda

Um Amazon-Kinesis-Data-Streams-Datensätze mit Lambda zu verarbeiten, erstellen Sie eine Lambda-Zuordnung von Ereignisquellen. Sie können eine Lambda-Ereignisfunktion einem Standard-Iterator oder einem erweiterten Fan-Out-Consumer zuordnen. Weitere Informationen finden Sie unter Abfragen und Stapeln von Streams.

Erstellen einer Zuordnung von Ereignisquellen für Kinesis

Um Ihre Lambda-Ereignisfunktion mit Datensätzen aus Ihrem Datenstrom aufzurufen, erstellen Sie eine Ereignisquellenzuordnung. Sie können mehrere Ereignisquellenzuordnungen erstellen, um dieselben Daten mit mehreren Lambda-Ereignisfunktionen zu verarbeiten oder um Elemente aus mehreren Datenströmen mit einer einzigen Funktion zu verarbeiten. Bei der Verarbeitung von Elementen aus mehreren Datenströmen enthält jeder Batch Datensätze aus nur einem einzigen Shard oder Stream.

Sie können Zuordnungen von Ereignisquellen konfigurieren, um Datensätze aus einem Stream in einem anderen AWS-Konto zu verarbeiten. Weitere Informationen hierzu finden Sie unter the section called “Cross-account Zuordnungen”.

Bevor Sie eine Ereignisquellenzuordnung erstellen, müssen Sie Ihrer Lambda-Ereignisfunktion die Erlaubnis erteilen, aus einem Kinesis-Datenstream zu lesen. Lambda benötigt die folgenden Berechtigungen zum Verwalten von Ressourcen, die zu Ihrem Kinesis-Datenstrom gehören:

- Kinesis: DescribeStream
- Kinese: DescribeStreamSummary
- Kinese: GetRecords
- Kinese: GetShardIterator
- Kinese: ListShards
- Kinese: SubscribeToShard

Die AWS verwaltete Richtlinie AWSLambdaKinesisExecutionRoleumfasst diese Berechtigungen. Fügen Sie diese verwaltete Richtlinie zu Ihrer Funktion hinzu, wie im folgenden Verfahren beschrieben.

> **Note**
>
> - Sie benötigen keine `kinesis:ListStreams`-Berechtigung, um Zuordnungen von Ereignisquellen für Kinesis zu erstellen und zu verwalten. Wenn Sie jedoch eine Ereignisquellenzuordnung in der Konsole erstellen und nicht über diese Berechtigung verfügen, können Sie keinen Kinesis-Stream aus einer Dropdownliste auswählen und die Konsole zeigt eine Fehlermeldung an. Um die Zuordnung von Ereignisquellen zu erstellen, müssen Sie den Amazon-Ressourcennamen (ARN) Ihres Streams manuell eingeben.
> - Lambda ruft die APIs `kinesis:GetRecords` und `kinesis:GetShardIterator` auf, wenn fehlgeschlagene Aufrufe erneut versucht werden.

AWS-Managementkonsole

So fügen Sie Ihrer Event-Funktion Kinesis-Berechtigungen hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
3. Wählen Sie im Bereich „Ausführungsrolle“ unter Rollenname den Link zur Ausführungsrolle Ihrer Funktion aus. Dieser Link öffnet die Seite für diese Rolle in der IAM-Konsole.
4. Unter Berechtigungsrichtlinien im Abschnitt Berechtigungen hinzufügen wählen Sie dann Richtlinien anfügen aus.
5. Geben Sie im Suchfeld **`AWSLambdaKinesisExecutionRole`** ein.

6. Aktivieren Sie das Kontrollkästchen neben der Richtlinie und wählen Sie Berechtigung hinzufügen.

AWS CLI

So fügen Sie Ihrer Event-Funktion Kinesis-Berechtigungen hinzu

- Führen Sie den folgenden CLI-Befehl aus, um die `AWSLambdaKinesisExecutionRole` Richtlinie zur Ausführungsrolle Ihrer Funktion hinzuzufügen:

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

So fügen Sie Ihrer Event-Funktion Kinesis-Berechtigungen hinzu

- Fügen Sie der Definition Ihrer Funktion die `Policies` Eigenschaft hinzu, wie im folgenden Beispiel gezeigt:

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

Nachdem Sie die erforderlichen Berechtigungen konfiguriert haben, erstellen Sie die Zuordnung von Ereignisquellen.

AWS-Managementkonsole

So erstellen Sie die Kinesis-Zuordnung von Ereignisquellen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion aus.

2. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).
3. Wählen Sie unter Trigger-Konfiguration als Quelle die Option Kinesis aus.
4. Wählen Sie den Kinesis-Stream, für den Sie die Zuordnung von Ereignisquellen erstellen möchten und optional einen Konsumenten Ihres Streams.
5. (Optional) Bearbeiten Sie die Batch-Größe, die Startposition und das Batch-Fenster für Ihre Zuordnung von Ereignisquellen.
6. Wählen Sie Hinzufügen aus.

Wenn Sie Ihre Ereignisquellenzuordnung von der Konsole aus erstellen, muss Ihre IAM-Rolle über die Berechtigungen kinesis: ListStreams und kinesis: verfügen. ListStreamConsumers

AWS CLI

So erstellen Sie die Kinesis-Zuordnung von Ereignisquellen

- Führen Sie den folgenden CLI-Befehl aus, um eine Kinesis-Zuordnung von Ereignisquellen zu erstellen. Wählen Sie Ihre eigene Batch-Größe und Startposition entsprechend Ihrem Anwendungsfall.

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

Um ein Batching-Fenster anzugeben, fügen Sie die `--maximum-batching-window-in-seconds`-Option hinzu. Weitere Informationen zur Verwendung dieses und anderer Parameter finden Sie unter create-event-source-mapping in der AWS CLI Befehlsreferenz.

AWS SAM

So erstellen Sie die Kinesis-Zuordnung von Ereignisquellen

- Fügen Sie der Definition Ihrer Funktion die `KinesisEvent` Eigenschaft hinzu, wie im folgenden Beispiel gezeigt:

```
Resources:
  MyFunction:
```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
      Events:
        KinesisEvent:
          Type: Kinesis
          Properties:
            Stream: !GetAtt MyKinesisStream.Arn
            StartingPosition: LATEST
            BatchSize: 100

  MyKinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1
```

Weitere Informationen zum Erstellen einer Ereignisquellenzuordnung für Kinesis Data Streams finden Sie unter Kinesis im AWS Serverless Application Model Entwicklerhandbuch. AWS SAM

Abfrage und Startposition des Streams

Beachten Sie, dass die Stream-Abfrage bei der Erstellung und Aktualisierung der Zuordnung von Ereignisquellen letztendlich konsistent ist.

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Bei Aktualisierungen der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass, wenn Sie `LATEST` als Startposition für den Stream angeben, die Zuordnung von Ereignisquellen bei der Erstellung oder Aktualisierung möglicherweise Ereignisse übersieht. Um sicherzustellen, dass keine Ereignisse übersehen werden, geben Sie die Startposition des Streams als `TRIM_HORIZON` oder `AT_TIMESTAMP` an.

Erstellen einer kontoübergreifenden Zuordnung von Ereignisquellen

Amazon Kinesis Data Streams unterstützt ressourcenbasierte Richtlinien. Aus diesem Grund können Sie Daten, die in einen Stream aufgenommen wurden, in einem Konto AWS-Konto mit einer Lambda-Ereignisfunktion in einem anderen Konto verarbeiten.

Um eine Ereignisquellenzuordnung für Ihre Lambda-Ereignisfunktion mithilfe eines Kinesis-Streams in einem anderen zu erstellen AWS-Konto, müssen Sie den Stream mithilfe einer ressourcenbasierten Richtlinie konfigurieren, um Ihrer Lambda-Ereignisfunktion die Berechtigung zum Lesen von Elementen zu erteilen. Informationen dazu, wie Sie Ihren Stream so konfigurieren, dass er kontoübergreifenden Zugriff ermöglicht, finden Sie unter Zugriff mit kontoübergreifenden AWS Lambda Funktionen teilen im Amazon Kinesis Streams Streams-Entwicklerhandbuch.

Nachdem Sie Ihren Stream mit einer ressourcenbasierten Richtlinie konfiguriert haben, die Ihrer Lambda-Ereignisfunktion die erforderlichen Berechtigungen erteilt, erstellen Sie die Ereignisquellenzuordnung mit einer der im vorherigen Abschnitt beschriebenen Methoden.

Wenn Sie Ihre Zuordnung von Ereignisquellen über die Lambda-Konsole erstellen möchten, fügen Sie den ARN Ihres Streams direkt in das Eingabefeld ein. Wenn Sie einen Verbraucher für Ihren Stream angeben möchten, wird durch Einfügen der ARN des Verbrauchers automatisch das Stream-Feld ausgefüllt.

Konfigurieren einer teilweisen Batch-Antwort mit Kinesis Data Streams und Lambda

Beim Konsumieren und Verarbeiten von Streaming-Daten aus einer Ereignisquelle werden standardmäßig Lambda-Checkpoints auf die höchste Sequenznummer eines Batches nur dann überprüft, wenn der Batch ein voller Erfolg ist. Lambda behandelt alle anderen Ergebnisse als einen vollständigen Fehler und versucht, den Batch bis zum Wiederholungslimit zu verarbeiten. Um beim Verarbeiten von Stapeln aus einem Stream Teilerfolge zu ermöglichen, aktivieren Sie `ReportBatchItemFailures`. Das Zulassen teilweiser Erfolge kann dazu beitragen, die Anzahl der Wiederholungen für einen Datensatz zu reduzieren, verhindert jedoch nicht vollständig die Möglichkeit von Wiederholungen in einem erfolgreichen Datensatz.

Zum Aktivieren von `ReportBatchItemFailures` fügen Sie den Enum-Wert **`ReportBatchItemFailures`** der FunctionResponseTypes-Liste hinzu. Diese Liste gibt an, welche Antworttypen für Ihre Event-Funktion aktiviert sind. Sie können diese Liste konfigurieren, wenn Sie eine Zuordnung von Ereignisquellen erstellen oder aktualisieren.

> **Note**
>
> Selbst wenn Ihr Funktionscode partielle Batch-Fehlerantworten zurückgibt, werden diese Antworten nicht von Lambda verarbeitet, es sei denn, die `ReportBatchItemFailures` Funktion ist explizit für Ihre Ereignisquellenzuordnung aktiviert.

Berichtsyntax

Beim Konfigurieren von Berichten zu Batch-Elementfehlern wird die `StreamsEventResponse`-Klasse mit einer Liste von Batch-Elementfehlern zurückgegeben. Sie können ein `StreamsEventResponse`-Objekt verwenden, um die Sequenznummer des ersten fehlgeschlagenen Datensatzes im Batch zurückzugeben. Sie können auch Ihre eigene benutzerdefinierte Klasse mit der richtigen Antwortsyntax erstellen. Die folgende JSON-Struktur zeigt die erforderliche Antwortsyntax:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> Wenn das `batchItemFailures`-Array mehrere Elemente enthält, verwendet Lambda den Datensatz mit der niedrigsten Sequenznummer als Kontrollpunkt. Lambda wiederholt dann alle Datensätze ab diesem Kontrollpunkt.

Erfolgs- und Misserfolgsbedingungen

Lambda behandelt einen Batch als vollständigen Erfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere `batchItemFailure`-Liste
- Eine ungültige `batchItemFailure`-Liste
- Ein leeres `EventResponse`
- Ein ungültiges `EventResponse`

Lambda behandelt einen Batch als vollständigen Misserfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere Zeichenfolge `itemIdentifier`
- Ein ungültiges `itemIdentifier`
- Ein `itemIdentifier` mit einem falschen Schlüsselnamen

Lambda wiederholt Fehler basierend auf Ihrer Wiederholungsstrategie.

Einen Batch halbieren

Wenn Ihr Aufruf fehlschlägt und `BisectBatchOnFunctionError` eingeschaltet ist, wird der Stapel unabhängig von Ihrer `ReportBatchItemFailures`-Einstellung halbiert.

Wenn eine partielle Batch-Erfolgsantwort empfangen wird und sowohl `BisectBatchOnFunctionError` als auch `ReportBatchItemFailures` aktiviert sind, wird der Batch mit der zurückgegebenen Sequenznummer halbiert und Lambda versucht nur die verbleibenden Datensätze erneut.

Um die Implementierung der partiellen Batch-Antwortlogik zu vereinfachen, sollten Sie in Erwägung ziehen, das Batch Processor Utility von Powertools for zu verwenden AWS Lambda, das diese Komplexität automatisch für Sie erledigt.

Hier sind einige Beispiele für Funktionscodes, die die Liste der fehlgeschlagenen Nachrichten-IDs im Batch zurückgeben:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
```

```
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
```

```
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
```

```
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Melden von Fehlern Kinesis Kinesis-Batch-Elementen mit Lambda unter Verwendung von. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";
```

```
const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();
```

```
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
```

```
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
```

```
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());
```

```
    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Verwenden von Powertools für AWS Lambda Batch-Prozessor

Das Batchverarbeitungsprogramm von Powertools for verarbeitet AWS Lambda automatisch die Logik für partielle Batch-Antworten und reduziert so die Komplexität der Implementierung von Batch-Fehlerberichten. Hier sind Beispiele für die Verwendung des Batch-Prozessors:

Python

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import KinesisEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.KinesisDataStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { KinesisEvent, Context } from 'aws-lambda';
```

```
const processor = new BatchProcessor(EventType.KinesisDataStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: KinesisEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class KinesisStreamBatchHandler implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<KinesisEvent, StreamsEventResponse> handler;

    public KinesisStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withKinesisBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
```

```
    }

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent kinesisEvent, Context
 context) {
        return handler.processBatch(kinesisEvent, context);
    }

    private void processMessage(KinesisEvent.KinesisEventRecord kinesisEventRecord,
 Context context) {
        // Process the stream record
    }
}
```

.NET

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;

[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class OrderEvent
{
    public string? OrderId { get; set; }
    public string? CustomerId { get; set; }
    public decimal Amount { get; set; }
```

```
    public DateTime OrderDate { get; set; }
}

internal class TypedKinesisRecordHandler : ITypedRecordHandler<OrderEvent>
{
    public async Task<RecordHandlerResult> HandleAsync(OrderEvent orderEvent,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(orderEvent.OrderId))
        {
            throw new ArgumentException("Order ID is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedKinesisRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(KinesisEvent _)
    {
        return TypedKinesisStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

Aufbewahrung verworfener Batch-Datensätze für eine Kinesis Data Streams-Ereignisquelle in Lambda

Die Fehlerbehandlung für Kinesis-Ereignisquellenzuordnungen hängt davon ab, ob der Fehler vor dem Aufrufen der Ereignisfunktion oder während des Funktionsaufrufs auftritt:

- Vor dem Aufruf: Wenn eine Lambda-Ereignisquellenzuordnung die Funktion aufgrund von Drosselung oder anderen Problemen nicht aufrufen kann, versucht sie es erneut, bis die Datensätze ablaufen oder das in der Ereignisquellenzuordnung () konfigurierte Höchstalter überschreiten. MaximumRecordAgeInSeconds
- Während des Aufrufs: Wenn die Funktion aufgerufen wird, aber einen Fehler zurückgibt, versucht Lambda es erneut, bis die Datensätze ablaufen, das Höchstalter (MaximumRecordAgeInSeconds) überschreiten oder die konfigurierte Wiederholungsquote () erreicht haben. MaximumRetryAttempts Bei Funktionsfehlern können Sie auch konfigurieren BisectBatchOnFunctionError, dass ein

fehlgeschlagener Batch in zwei kleinere Batches aufgeteilt wird, wodurch fehlerhafte Datensätze isoliert und Timeouts vermieden werden. Durch das Aufteilen von Batches wird die Quote für Wiederholungen nicht verbraucht.

Wenn die Fehlerbehandlungsmaßnahmen fehlschlagen, verwirft Lambda die Datensätze und setzt die Verarbeitung von Batches aus dem Stream fort. Bei den Standardeinstellungen bedeutet dies, dass ein fehlerhafter Datensatz die Verarbeitung auf dem betroffenen Shard für bis zu eine Woche blockieren kann. Um dies zu vermeiden, konfigurieren Sie die Ereignisquellenzuordnung Ihrer Funktion mit einer angemessenen Anzahl von Wiederholungen und einem maximalen Datensatzalter, das zu Ihrem Anwendungsfall passt.

Konfigurieren von Zielen für fehlgeschlagene Aufrufe

Um Aufzeichnungen über fehlgeschlagene Aufrufe der Ereignisquellenzuordnung beizubehalten, fügen Sie der Ereignisquellenzuordnung Ihrer Ereignisfunktion ein Ziel hinzu. Jeder an das Ziel gesendete Datensatz ist ein JSON-Dokument mit Metadaten über den fehlgeschlagenen Aufruf. Bei Amazon S3-Zielen sendet Lambda auch den gesamten Aufrufdatensatz zusammen mit den Metadaten. Sie können jedes Amazon SNS SNS-Thema, jede Amazon SQS SQS-Warteschlange, jeden Amazon S3 S3-Bucket oder Kafka als Ziel konfigurieren.

Bei Amazon-S3-Zielen können Sie das Feature Amazon S3 Event Notifications verwenden, um Benachrichtigungen zu erhalten, wenn Objekte in Ihren S3-Ziel-Bucket hochgeladen werden. Sie können S3-Ereignisbenachrichtigungen auch so konfigurieren, dass eine weitere Lambda-Ereignisfunktion aufgerufen wird, um die automatische Verarbeitung fehlgeschlagener Batches durchzuführen.

Ihre Ausführungsrolle muss über Berechtigungen für das Ziel verfügen:

- Für ein SQS-Ziel: sqs: SendMessage
- Für ein SNS-Ziel: sns:Publish
- Für ein S3-Ziel: s3: und s3: PutObject ListBucket
- Für ein Kafka-Ziel: kafka-cluster: WriteData

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das

angegebene Kafka-Thema. Weitere Informationen finden Sie unter the section called “Kafka-Ziel bei einem Ausfall”.

Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für ein S3-Ziel aktiviert haben, muss die Ausführungsrolle Ihrer Funktion auch die Berechtigung haben, kms: aufzurufen. GenerateDataKey Wenn sich der KMS-Schlüssel und das S3-Bucket-Ziel in einem anderen Konto befinden als Ihre Lambda-Ereignisfunktion und Ausführungsrolle, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, um kms: GenerateDataKey zuzulassen.

Gehen Sie folgendermaßen vor, um ein Ausfallziel mit der Konsole zu konfigurieren:

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie als Quelle die Option Aufruf der Zuordnung von Ereignisquellen aus.
5. Wählen Sie für die Zuordnung von Ereignisquellen eine Ereignisquelle aus, die für diese Funktion konfiguriert ist.
6. Wählen Sie für Bedingung die Option Bei Ausfall aus. Für Aufrufe zur Zuordnung von Ereignisquellen ist dies die einzig akzeptierte Bedingung.
7. Wählen Sie unter Zieltyp den Zieltyp aus, an den Lambda Aufrufdatensätze sendet.
8. Wählen Sie unter Destination (Ziel) eine Ressource aus.
9. Wählen Sie Speichern.

Sie können mit AWS Command Line Interface (AWS CLI) auch ein Ziel für den Fall eines Fehlers konfigurieren. Mit dem folgenden Befehl create-event-source-mapping wird beispielsweise eine Zuordnung von Ereignisquellen mit einem SQS-Ziel zu `MyFunction` hinzugefügt:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

Der folgende update-event-source-mapping-Befehl aktualisiert eine Zuordnung von Ereignisquellen, um fehlgeschlagene Aufrufdatensätze nach zwei Wiederholungsversuchen an ein SNS-Ziel zu senden oder wenn die Datensätze älter als eine Stunde sind.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

Aktualisierte Einstellungen werden asynchron angewendet und werden erst nach Abschluss des Vorgangs in der Ausgabe berücksichtigt. Verwenden Sie den get-event-source-mapping-Befehl, um den aktuellen Status anzuzeigen.

Um ein Ziel zu entfernen, geben Sie eine leere Zeichenfolge als Argument für den `destination-config`-Parameter an:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Bewährte Methoden für die Sicherheit in Amazon S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze aus Ihrer Event-Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Lambda-Ereignisfunktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:

    a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.

    b. Wählen Sie im Richtlinien-Editor JSON aus.

c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.

d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.

e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle (CLI) einer Event-Funktion eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Beispiel für einen Amazon SNS- und Amazon SQS-Aufrufsatz

Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kinesis-Ereignisquelle an eine SQS-Warteschlange oder ein SNS-Thema sendet. Da Lambda nur die Metadaten für diese Zieltypen sendet, verwenden Sie die Felder `streamArn`, `shardId`, `startSequenceNumber` und `endSequenceNumber`, um den vollständigen Originaldatensatz abzurufen. Alle in der `KinesisBatchInfo`-Eigenschaft angezeigten Felder sind immer vorhanden.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
```

```
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

Sie können diese Informationen verwenden, um die betroffenen Datensätze aus dem Stream für die Fehlersuche abzurufen. Die tatsächlichen Datensätze sind nicht enthalten, daher müssen Sie diesen Datensatz verarbeiten und aus dem Stream abrufen, bevor sie ablaufen und verloren gehen.

Beispiel eines Amazon S3-Aufrufdatensatzes

Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kinesis-Ereignisquelle an einen Amazon S3-Bucket sendet. Zusätzlich zu allen Feldern aus dem vorherigen Beispiel für SQS- und SNS-Ziele enthält das Feld `payload` den ursprünglichen Aufrufdatensatz als maskierte JSON-Zeichenfolge.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
```

```
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

Das S3-Objekt, das den Aufrufdatensatz enthält, verwendet die folgende Namenskonvention:

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implementierung der zustandsabhängigen Verarbeitung von Kinesis-Datenströmen in Lambda

Lambda Event Functions können kontinuierliche Stream-Verarbeitungsanwendungen ausführen. Ein Stream entspricht einer unbegrenzten Menge von Daten, die kontinuierlich durch Ihre Anwendung fließen. Um Informationen aus dieser sich ständig aktualisierenden Eingabe zu analysieren, können Sie die enthaltenen Datensätze mithilfe eines zeitlich definierten Fensters binden.

Rollierende Fenster sind unterschiedliche Zeitfenster, die sich in regelmäßigen Abständen öffnen und schließen. Standardmäßig sind Lambda-Aufrufe zustandslos – Sie können sie nicht für die Verarbeitung von Daten über mehrere kontinuierliche Aufrufe hinweg ohne eine externe Datenbank verwenden. Mit rollierenden Fenstern können Sie jedoch Ihren Status über Aufrufe hinweg beibehalten.

Dieser Zustand enthält das Gesamtergebnis der Nachrichten, die zuvor für das aktuelle Fenster verarbeitet wurden. Ihr Status kann für jeden Shard maximal 1 MB betragen. Wenn er diese Größe überschreitet, wird Lambda das Fenster vorzeitig beenden.

Jeder Datensatz in einem Stream gehört zu einem bestimmten Fenster. Lambda verarbeitet jeden Datensatz mindestens einmal, garantiert jedoch nicht, dass jeder Datensatz nur einmal verarbeitet wird. In seltenen Fällen, etwa bei der Fehlerbehandlung, werden einige Datensätze möglicherweise

mehrmals verarbeitet. Datensätze werden beim ersten Mal immer in der richtigen Reihenfolge verarbeitet. Wenn Datensätze mehr als einmal verarbeitet werden, werden sie nicht in der richtigen Reihenfolge verarbeitet.

Aggregation und Verarbeitung

Ihre benutzerverwaltete Funktion wird sowohl zur Aggregation als auch zur Verarbeitung der Endergebnisse dieser Aggregation aufgerufen. Lambda aggregiert alle im Fenster empfangenen Datensätze. Sie können diese Datensätze in mehreren Stapeln erhalten, jeweils als ein separater Aufruf. Jeder Aufruf erhält einen Zustand.

Wenn Sie Tumbling-Fenster verwenden, muss Ihre Antwort auf die Lambda-Ereignisfunktion daher eine `state` Eigenschaft enthalten. Wenn die Antwort keine `state`-Eigenschaft enthält, betrachtet Lambda dies als fehlgeschlagenen Aufruf. Um diese Bedingung zu erfüllen, kann Ihre Funktion ein `TimeWindowEventResponse`-Objekt zurückgeben, das die folgende JSON-Form aufweist:

Example **`TimeWindowEventResponse`**-Werte

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Für Java-Funktionen empfehlen wir, eine `Map<String, String>` zu verwenden, um den Status darzustellen.

Am Ende des Fensters `isFinalInvokeForWindow` wird das Flag so gesetzt, dass es `true` sich um den endgültigen Status handelt und dass er zur Verarbeitung bereit ist. Nach der Verarbeitung werden das Fenster und Ihr endgültiger Aufruf wird abgeschlossen, und dann wird der Zustand gelöscht.

Am Ende Ihres Fensters verwendet Lambda die endgültige Verarbeitung für Aktionen an den Aggregationsergebnissen. Ihre endgültige Verarbeitung wird synchron aufgerufen. Nach

erfolgreichem Aufruf zeigt Ihre Funktion auf die Sequenznummer und die Stream-Verarbeitung wird fortgesetzt. Wenn der Aufruf nicht erfolgreich ist, unterbricht Ihre Lambda-Ereignisfunktion die weitere Verarbeitung bis zu einem erfolgreichen Aufruf.

Example KinesisTimeWindowEvent


```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Konfiguration

Sie können rollierende Fenster konfigurieren, wenn Sie eine Ereignisquellenzuordnung erstellen oder aktualisieren. Um ein Taumelfenster zu konfigurieren, geben Sie das Fenster in Sekunden () an. TumblingWindowInSeconds Mit dem folgenden Beispielbefehl AWS Command Line Interface (AWS CLI) wird eine Quellenzuordnung für Streaming-Ereignisse erstellt, die ein Wechselfenster von 120 Sekunden hat. Die für die Aggregation und Verarbeitung definierte Lambda-Ereignisfunktion trägt den Namen. `tumbling-window-example-function`

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda bestimmt die rollierenden Fenstergrenzen basierend auf dem Zeitpunkt, zu dem Datensätze in den Stream eingefügt wurden. Für alle Datensätze steht ein ungefährer Zeitstempel zur Verfügung, den Lambda in Grenzbestimmungen verwendet.

Rollierende Fensteraggregationen unterstützen kein Resharding. Wenn ein Shard endet, betrachtet Lambda das aktuelle Fenster als geschlossen, und alle untergeordneten Shards beginnen ihr eigenes Fenster in einem neuen Zustand. Wenn dem aktuellen Fenster keine neuen Datensätze hinzugefügt werden, wartet Lambda bis zu 2 Minuten, bevor es annimmt, dass das Fenster zu Ende ist. Dadurch wird sichergestellt, dass die Funktion alle Datensätze im aktuellen Fenster liest, auch wenn die Datensätze in Abständen hinzugefügt werden.

Rollierende Fenster unterstützen vollständig die bestehenden Wiederholungsrichtlinien `maxRetryAttempts` und `maxRecordAge`.

Example Handler.py — Aggregation und Verarbeitung

Die folgende Python-Funktion veranschaulicht, wie Sie Ihren Endzustand aggregieren und dann verarbeiten:

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
```

```
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Lambda-Parameter für Amazon Kinesis Data Streams Zuordnungen von Ereignisquellen

Alle Lambda-Ereignisquellenzuordnungen verwenden dieselben API-Operationen CreateEventSourceMapping. UpdateEventSourceMapping Allerdings gelten nur einige der Parameter für Kinesis.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
BisectBatchOnFunctionError	N	false	Keine
DestinationConfig	N	N/A	Ein Ziel der Amazon-SQS-Warteschlange oder des Amazon-SNS-Themas für verworfene Datensätze. Weitere Informationen finden Sie unter Konfigurieren von

Parameter	Erforderlich	Standard	Hinweise
			Zielen für fehlgeschlagene Aufrufe.
Aktiviert	N	true	Keine
EventSourceArn	Y	N/A	Der ARN des Datenstroms oder eines Stream-Konsumenten
FunctionName	Y	N/A	Keine
FunctionResponseTypes	N	N/A	Damit Ihre Event-Funktion bestimmte Fehler in einem Batch melden kann, fügen Sie den Wert in ein. `ReportBatchItemFailures FunctionResponseTypes` Weitere Informationen finden Sie unter Konfigurieren einer teilweisen Batch-Antwort mit Kinesis Data Streams und Lambda.
MaximumBatchingWindowInSeconds	N	0	Keine

Parameter	Erforderlich	Standard	Hinweise
MaximumRecordAgeInSeconds	N	-1	-1 bedeutet unendlich: Lambda verwirft keine Datensätze (Kinesis Data Streams-Datenaufbewahrungseinstellungen gelten weiterhin) Minimum: -1 Höchstwert: 604 800
MaximumRetryAttempts	N	-1	-1 bedeutet unendlich: Fehlgeschlagene Datensätze werden wiederholt, bis der Datensatz abläuft Minimum: -1 Höchstwert: 10 000.
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON, oder LATEST
StartingPositionTimestamp	N	N/A	Nur gültig, wenn der Wert auf AT_TIMESTAMP gesetzt StartingPosition ist. Die Zeit, ab der mit dem Lesen begonnen werden soll, in Unix-Zeitsekunden

Parameter	Erforderlich	Standard	Hinweise
TumblingWindowInSeconds	N	N/A	Minimum: 0 Maximum: 900

Verwendung der Ereignisfilterung mit einer Kinesis-Ereignisquelle

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Event-Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für Kinesis-Ereignisquellen.

> **Note**
>
> Kinesis-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `data`-Schlüssel.

Themen

- Grundlagen der Kinesis-Ereignisfilterung
- Filtern von aggregierten Kinesis-Datensätzen

Grundlagen der Kinesis-Ereignisfilterung

Angenommen, ein Producer gibt JSON-formatierte Daten in Ihren Kinesis-Datenstrom ein. Ein Beispieldatensatz würde wie folgt aussehen, wobei die JSON-Daten im `data`-Feld in eine Base64-kodierte Zeichenfolge umgewandelt wurden.

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
```

```
        },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

Solange die Daten, die der Producer in den Stream einspeist, als JSON gültig sind, können Sie die Ereignisfilterung verwenden, um Datensätze anhand des `data`-Schlüssels zu filtern. Nehmen wir an, ein Produzent gibt Datensätze im folgenden JSON-Format in Ihren Kinesis-Stream ein.

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

Um nur die Datensätze zu filtern, bei denen der Ordertyp „Kaufen“ lautet, würde das `FilterCriteria` Objekt wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
```

```
            }
        }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer AWS SAM Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
```

```
Filters:
  - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

Um Ereignisse aus Kinesis-Quellen ordnungsgemäß zu filtern, müssen sowohl das Datenfeld als auch die Filterkriterien für das Datenfeld ein gültiges JSON-Format aufweisen. Wenn eines der Felder kein gültiges JSON-Format hat, verwirft Lambda die Nachricht oder gibt eine Ausnahme aus. In der folgenden Tabelle ist das Verhalten zusammengefasst:

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Non-JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuordnung eine Ausnahme aus. Das Filtermuster für Dateneigenschaften muss ein gültiges JSON-Format haben.
Non-JSON	Gültiges JSON	Lambda verwirft den Datensatz.
Non-JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Non-JSON	Non-JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuo

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
		rdnung eine Ausnahme aus. Das Filtermuster für Dateneige nschaften muss ein gültiges JSON-Format haben.

Filtern von aggregierten Kinesis-Datensätzen

Mit Kinesis können Sie mehrere Datensätze in einem einzigen Kinesis-Datenstrom-Datensatz zusammenfassen, um Ihren Datendurchsatz zu erhöhen. Lambda kann Filterkriterien nur auf aggregierte Datensätze anwenden, wenn Sie Kinesis Enhanced Fanout verwenden. Das Filtern aggregierter Datensätze mit Standard-Kinesis wird nicht unterstützt. Wenn Sie Enhanced Fan-Out verwenden, konfigurieren Sie einen Dediced-Throughput-Consumer von Kinesis, der als Auslöser für Ihre Lambda Event Function fungiert. Lambda filtert dann die aggregierten Datensätze und übergibt nur die Datensätze, die Ihren Filterkriterien entsprechen.

Weitere Informationen zur Aggregation von Kinesis-Datensätzen finden Sie im Abschnitt Aggregation auf der Seite mit den wichtigsten Konzepten der Kinesis Producer Library (KPL). Weitere Informationen zur Verwendung von Lambda mit Kinesis Enhanced Fan-Out finden Sie im Compute-Blog unter Steigerung der Echtzeit-Stream-Verarbeitungsleistung mit Amazon Kinesis Data Streams Enhanced Fan-Out und AWS Lambda. AWS

Anleitung: Verwenden von Lambda mit Kinesis Data Streams

In diesem Tutorial erstellen Sie eine Lambda-Ereignisfunktion, um Ereignisse aus einem Amazon Kinesis Kinesis-Datenstream zu verarbeiten.

1. Die benutzerdefinierte Anwendung schreibt die Datensätze zum Stream.
2. AWS Lambda fragt den Stream ab und ruft, wenn es neue Datensätze im Stream erkennt, Ihre Lambda-Event-Funktion auf.
3. AWS Lambda führt die Lambda-Ereignisfunktion aus, indem sie die Ausführungsrolle annimmt, die Sie bei der Erstellung der Lambda-Ereignisfunktion angegeben haben.

Voraussetzungen

Installieren Sie die AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle

Erstellen Sie die Ausführungsrolle, die Ihrer Funktion die Berechtigung zum Zugriff auf AWS Ressourcen erteilt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Trusted entity (Vertrauenswürdige Entität – AWS Lambda.
   - Berechtigungen — AWSLambdaKinesisExecutionRole.
   - Role name (Name der Rolle – **`lambda-kinesis-role`**.

Die AWSLambdaKinesisExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion zum Lesen von Elementen aus Kinesis und zum CloudWatch Schreiben von Protokollen in Logs benötigt.

Erstellen Sie die Event-Funktion

Erstellen Sie eine Lambda-Ereignisfunktion, die Ihre Kinesis-Nachrichten verarbeitet. Der Funktionscode protokolliert die Ereignis-ID und die Ereignisdaten des Kinesis-Datensatzes in CloudWatch Logs.

In diesem Tutorial wird die Node.js 24-Runtime verwendet, aber wir haben auch Beispielcode in anderen Runtime-Sprachen bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um Code für die gewünschte Laufzeit anzusehen. Der JavaScript Code, den Sie in diesem Schritt verwenden, befindet sich im ersten Beispiel, das auf der JavaScriptRegisterkarte angezeigt wird.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
```

```
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
```

```
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
```

```
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Kinesis-Ereignis mit Lambda verwenden. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
```

```
  return data;
}
```

Ein Kinesis-Ereignis mit Lambda verwenden. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
```

```
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
```

```
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
```

```
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
```

```
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
```

```
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Um die Event-Funktion zu erstellen

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir kinesis-tutorial
   cd kinesis-tutorial
   ```

2. Kopieren Sie den JavaScript Beispielcode in eine neue Datei mit dem Namen`index.js`.

3. Erstellen Sie ein Bereitstellungspaket.

   ```
   zip function.zip index.js
   ```

4. Erstellen Sie mit dem `create-function` Befehl eine Lambda-Ereignisfunktion.

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

Testen Sie die Lambda-Ereignisfunktion

Rufen Sie Ihre Lambda-Ereignisfunktion manuell mit dem `invoke` AWS Lambda CLI-Befehl und einem Kinesis-Beispielereignis auf.

Um die Lambda-Event-Funktion zu testen

1. Kopieren Sie das folgende JSON in eine Datei und speichern Sie sie unter `input.txt`.

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
}
```

2. Verwenden Sie den `invoke`-Befehl, um das Ereignis an die Funktion zu senden.

```
aws lambda invoke --function-name ProcessKinesisRecords \
```

```
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Antwort wird in gespeicher `out.txt`.

Kinesis-Stream erstellen

Verwenden Sie den Befehl `create-stream` , um einen Stream zu erstellen.

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

Führen Sie den folgenden `describe-stream`-Befehl aus, um den Stream-ARN zu erhalten.

```
aws kinesis describe-stream --stream-name lambda-stream
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
```

```
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

Im nächsten Schritt verwenden Sie den Stream-ARN, um den Stream mit Ihrer Lambda-Ereignisfunktion zu verknüpfen.

Fügen Sie eine Ereignisquelle hinzu in AWS Lambda

Führen Sie den Befehl AWS CLI `add-event-source` aus.

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

Notieren Sie die Zuweisungs-ID zur späteren Verwendung. Sie erhalten eine Liste der Zuweisungen von Ereignisquellen, indem Sie den `list-event-source-mappings`-Befehl ausführen.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

In der Antwort können Sie überprüfen, ob der Statuswert is `enabled`. Ereignisquellen-Zuweisungen können deaktiviert werden, um Abfragen vorübergehend zu pausieren, ohne dass Datensätze verloren gehen.

Testen der Einrichtung

Zum Testen der Ereignisquellen-Zuweisung fügen Sie Ihrem Kinesis-Stream Ereignisdatensätze hinzu. Der `--data` Wert ist eine Zeichenfolge, die die CLI in Base64 codiert, bevor sie an Kinesis gesendet wird. Der gleiche Befehl kann mehrmals ausgeführt werden, um dem Stream mehrere Datensätze hinzuzufügen.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda verwendet die Ausführungsrolle, um Datensätze aus dem Stream zu lesen. Dann ruft es Ihre Lambda-Ereignisfunktion auf und übergibt Stapel von Datensätzen. Die Funktion dekodiert Daten aus jedem Datensatz und protokolliert sie, wobei die Ausgabe an Logs gesendet wird. CloudWatch Zeigen Sie die Protokolle in der CloudWatch -Konsole an.

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie den Kinesis-Stream

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die Kinesis-Konsole unter https://console.aws.amazon.com/kinesis.
2. Wählen Sie den von Ihnen erstellten Stream aus
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`delete`** in das Texteingabefeld ein.

5. Wählen Sie Löschen aus.

Verwendung von Lambda mit Kubernetes

Sie können Lambda Event Functions mit der Kubernetes-API mithilfe von AWS Controllers for Kubernetes (ACK) oder Crossplane bereitstellen und verwalten.

AWS Controller für Kubernetes (ACK)

Sie können ACK verwenden, um AWS Ressourcen über die Kubernetes-API bereitzustellen und zu verwalten. AWS Stellt über ACK benutzerdefinierte Open-Source-Controller für AWS Dienste wie Lambda, Amazon Elastic Container Registry (Amazon ECR), Amazon Simple Storage Service (Amazon S3) und Amazon AI bereit. SageMaker Jeder unterstützte AWS Service hat seinen eigenen benutzerdefinierten Controller. Installieren Sie in Ihrem Kubernetes-Cluster einen Controller für jeden AWS Dienst, den Sie verwenden möchten. Erstellen Sie anschließend eine benutzerdefinierte Ressourcendefinition (CRD), um die Ressourcen zu definieren. AWS

Wir empfehlen die Verwendung von Helm 3.8 oder höher, um ACK-Controller zu installieren. Jeder ACK-Controller verfügt über ein eigenes Helm-Chart, das den Controller, CRDs und Kubernetes RBAC-Regeln installiert. Weitere Informationen finden Sie unter Installieren eines ACK-Controllers in der ACK-Dokumentation.

Nachdem Sie die benutzerdefinierte ACK-Ressource erstellt haben, können Sie sie wie jedes andere integrierte Kubernetes-Objekt verwenden. Sie können Lambda Event Functions beispielsweise mit Ihren bevorzugten Kubernetes-Toolchains, einschließlich kubectl, bereitstellen und verwalten.

Hier sind einige Anwendungsbeispiele für die Bereitstellung von Lambda Event Functions über ACK:

- Ihr Unternehmen verwendet rollenbasierte Zugriffssteuerung (RBAC) und IAM-Rollen für Service-Konten, um Berechtigungsgrenzen zu erstellen. Mit ACK können Sie dieses Sicherheitsmodell für Lambda wiederverwenden, ohne neue Benutzer und Richtlinien erstellen zu müssen.
- Ihre Organisation verfügt über einen DevOps Prozess zur Bereitstellung von Ressourcen in einem Amazon Elastic Kubernetes Service (Amazon EKS) -Cluster mithilfe von Kubernetes-Manifesten. Mit ACK können Sie ein Manifest verwenden, um Lambda-Ereignisfunktionen bereitzustellen, ohne eine separate Infrastruktur als Codevorlagen erstellen zu müssen.

Weitere Informationen zur Verwendung von ACK finden Sie im Lambda-Tutorial in der ACK-Dokumentation.

Crossplane

Crossplane ist ein Open-Source-Projekt der Cloud Native Computing Foundation (CNCF), das Kubernetes zur Verwaltung von Cloud-Infrastrukturressourcen verwendet. Mit Crossplane können Entwickler Infrastruktur anfordern, ohne deren Komplexität verstehen zu müssen. Plattformteams behalten die Kontrolle über die Bereitstellung und Verwaltung der Infrastruktur.

Mit Crossplane können Sie Lambda Event Functions mit Ihren bevorzugten Kubernetes-Toolchains wie kubectl und jeder Pipeline, die Manifeste für Kubernetes bereitstellen kann, bereitstellen und verwalten. CI/CD Hier sind einige Anwendungsbeispiele für die Bereitstellung von Lambda Event Functions über Crossplane:

- Ihr Unternehmen möchte die Einhaltung der Vorschriften durchsetzen, indem es sicherstellt, dass Lambda Event Functions über die richtigen Tags verfügen. Plattformteams können Crossplane Compositions verwenden, um diese Richtlinie durch API-Abstraktionen zu definieren. Entwickler können diese Abstraktionen dann verwenden, um Lambda-Ereignisfunktionen mit Tags bereitzustellen.
- Ihr Projekt verwendet Kubernetes GitOps . In diesem Modell gleicht Kubernetes kontinuierlich das Git-Repository (gewünschter Status) mit den im Cluster ausgeführten Ressourcen (aktueller Status) ab. Wenn es Unterschiede gibt, nimmt der GitOps Prozess automatisch Änderungen am Cluster vor. Sie können es GitOps zusammen mit Kubernetes für die Bereitstellung und Verwaltung von Lambda Event Functions über Crossplane verwenden und dabei vertraute Kubernetes-Tools und -Konzepte wie CRDs und Controller verwenden.

Weitere Informationen zur Verwendung von Crossplane mit Lambda finden Sie hier:

- AWS Blueprints for Crossplane: Dieses Repository enthält Beispiele für die Verwendung von Crossplane zur Bereitstellung AWS von Ressourcen, einschließlich Lambda Event Functions.

  > **Note**
  >
  > AWS Blueprints für Crossplane befinden sich in der aktiven Entwicklung und sollten nicht in der Produktion verwendet werden.

- Bereitstellung von Lambda mit Amazon EKS und Crossplane: Dieses Video zeigt ein fortgeschrittenes Beispiel für die Implementierung einer AWS serverlosen Architektur mit Crossplane und untersucht das Design sowohl aus Entwickler- als auch aus Plattformperspektive.

Verwenden von Lambda mit Amazon MQ

> **Note**
>
> Wenn Sie Daten an ein anderes Ziel als eine Lambda-Ereignisfunktion senden oder die Daten vor dem Senden anreichern möchten, finden Sie weitere Informationen unter Amazon EventBridge Pipes.

Amazon MQ ist ein verwalteter Message-Broker-Service für Apache ActiveMQ und RabbitMQ. Mit einem Message Broker können Software-Anwendungen und -Komponenten mithilfe verschiedener Programmiersprachen, Betriebssysteme und formeller Messaging-Protokolle entweder über Themen- oder Queue-Ereigniszielen miteinander kommunizieren.

Amazon MQ kann auch Amazon-Elastic-Compute-Cloud-(Amazon-EC2)-Instances in Ihrem Namen verwalten, indem es ActiveMQ- oder RabbitMQ-Broker installiert und verschiedene Netzwerktopologien und andere Infrastrukturanforderungen bereitstellt.

Sie können eine Lambda-Ereignisfunktion verwenden, um Datensätze von Ihrem Amazon MQ Message Broker zu verarbeiten. Lambda ruft Ihre Funktion über ein Ereignisquellen-Zuweisung auf, eine Lambda-Ressource, die Nachrichten von Ihrem Broker liest und die Funktion synchron aufruft. Informationen zu den Konfigurationsoptionen finden Sie unter. the section called “Parameter” Informationen zu häufigen Fehlern finden Sie unterthe section called “Fehlerbehebung”.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Das Amazon-MQ-Ereignisquellen-Zuweisung hat die folgenden Konfigurationseinschränkungen:

- Parallelität — Lambda-Ereignisfunktionen, die eine Amazon MQ MQ-Ereignisquellenzuordnung verwenden, haben eine Standardeinstellung für maximale Parallelität. Für ActiveMQ begrenzt der Lambda-Service die Anzahl der gleichzeitigen Ausführungsumgebungen auf fünf pro

Amazon MQ-Zuordnung von Ereignisquellen. Für RabbitMQ ist die Anzahl der gleichzeitigen Ausführungsumgebungen auf 1 pro Amazon MQ-Zuordnung von Ereignisquellen begrenzt.

Selbst wenn Sie die reservierten oder bereitgestellten Parallelitätseinstellungen Ihrer Funktion ändern, stellt der Lambda-Service keine weiteren Ausführungsumgebungen zur Verfügung. Um eine Erhöhung der standardmäßigen maximalen Parallelität für eine einzelne Amazon MQ MQ-Ereignisquellenzuordnung zu beantragen, wenden Sie sich an Support die UUID der Ereignisquellenzuweisung sowie an die Region. Da Erhöhungen auf der Ebene der spezifischen Zuordnung von Ereignisquellen und nicht auf der Ebene des Kontos oder der Region vorgenommen werden, müssen Sie für jede Zuordnung von Ereignisquellen manuell eine Skalierungserhöhung anfordern.

- Kontoübergreifend – Lambda unterstützt keine kontoübergreifende Verarbeitung. Sie können Lambda nicht zur Verarbeitung von Datensätzen von einem Amazon-MQ-Message Broker verwenden, der sich in einem anderen AWS-Konto befindet.
- Authentifizierung — Für ActiveMQ wird nur SimpleAuthenticationPluginActiveMQ unterstützt. Für RabbitMQ wird nur der PLAIN-Authentifizierungsmechanismus unterstützt. Benutzer müssen es verwenden, AWS Secrets Manager um ihre Anmeldeinformationen zu verwalten. Weitere Informationen zur ActiveMQ-Authentifizierung finden Sie unter Integrieren von ActiveMQ-Brokern mit LDAP im Amazon-MQ-Entwicklerhandbuch.
- Verbindungskontingent – Broker haben eine maximale Anzahl zulässiger Verbindungen pro Wire-Level-Protokoll. Dieses Kontingent basiert auf dem Instance-Typ des Brokers. Weitere Informationen finden Sie im Abschnitt Broker in Kontingente in Amazon MQ im Amazon-MQ-Entwicklerhandbuch.
- Konnektivität – Sie können Broker in einer öffentlichen oder privaten Virtual Private Cloud (VPC) erstellen. Für private VPCs benötigt Ihre Lambda Event Function Zugriff auf die VPC, um Nachrichten zu empfangen. Weitere Informationen finden Sie unter the section called “Konfigurieren Sie die Netzwerksicherheit”, weiter unten in diesem Abschnitt.
- Ereignisziele – Es werden nur Warteschlangenziele unterstützt. Sie können jedoch ein virtuelles Thema verwenden, das sich intern wie ein Thema verhält, während es mit Lambda als eine Warteschlange interagiert. Weitere Informationen finden Sie unter virtuelle Ziele auf der Apache-ActiveMQ-Website und virtuelle Hosts auf der RabbitMQ-Website.
- Netzwerktopologie – Für ActiveMQ wird nur ein Einzelinstance- oder Standby-Broker pro Ereignisquellen-Zuweisung unterstützt. Für RabbitMQ wird pro Zuordnung der Ereignisquelle nur eine Broker- oder Clusterbereitstellung mit einer einzigen Instanz unterstützt. Single-instance Broker benötigen einen Failover-Endpunkt. Weitere Informationen zu diesen Broker-

Bereitstellungsmodi finden Sie unter Aktive MQ-Broker-Architektur und Broker-Architektur von Rabbit MQ im Entwicklerhandbuch für Amazon MQ.

- Protokolle – Die unterstützten Protokolle hängen vom Typ der Amazon-MQ-Integration ab.
  - Für ActiveMQ-Integrationen verwendet Lambda Nachrichten mithilfe des OpenWire/Java Message Service (JMS) -Protokolls. Es werden keine anderen Protokolle unterstützt, um Nachrichten zu verbrauchen. Innerhalb des JMS-Protokolls werden nur `TextMessage` und `BytesMessage` unterstützt. Lambda unterstützt auch benutzerdefinierte JMS-Eigenschaften. Weitere Informationen zum OpenWire Protokoll finden Sie OpenWireauf der Apache ActiveMQ-Website.
  - Bei RabbitMQ-Integrationen verbraucht Lambda Nachrichten mit dem AMQP-0-9-1-Protokoll. Es werden keine anderen Protokolle unterstützt, um Nachrichten zu verbrauchen. Weitere Informationen zur Implementierung des AMQP 0-9-1-Protokolls durch RabbitMQ finden Sie im Kompletten AMQ-0-9-1-Referenzhandbuch auf der RabbitMQ-Website.

Lambda unterstützt automatisch die neuesten Versionen von ActiveMQ und RabbitMQ, die Amazon MQ unterstützt. Die neuesten unterstützten Versionen finden Sie in den Amazon-MQ-Versionshinweisen im Amazon-MQ-Entwicklerhandbuch.

> **Note**
>
> Amazon MQ hat standardmäßig ein wöchentliches Wartungsfenster für Broker. Während dieses Zeitfensters sind Broker nicht verfügbar. Für Broker ohne Standby kann Lambda während dieses Fensters keine Nachrichten verarbeiten.

Themen

- Verstehen der Lambda-Verbrauchergruppe für Amazon MQ
- Konfiguration der Amazon MQ-Ereignisquelle für Lambda
- Parameter für die Zuordnung von Ereignisquellen
- Ereignisse aus einer Amazon MQ-Ereignisquelle filtern
- Beheben Sie Fehler Amazon MQ-Zuordnung von Ereignisquellen

Verstehen der Lambda-Verbrauchergruppe für Amazon MQ

Um mit Amazon MQ zu interagieren, erstellt Lambda eine Verbrauchergruppe, die aus Ihren Amazon-MQ-Brokern lesen kann. Die Verbrauchergruppe wird mit derselben ID wie die Ereignisquellen-Zuweisung-UUID erstellt.

Für Amazon-MQ-Ereignisquellen batcht Lambda Datensätze und sendet sie in einer einzigen Nutzlast an Ihre Funktion. Um das Verhalten zu steuern, können Sie das Batch-Fenster und die Batch-Größe konfigurieren. Lambda pullt Nachrichten, bis es die Nutzlastgröße von maximal 6 MB verarbeitet, das Batch-Fenster abläuft oder die Anzahl der Datensätze die volle Batch-Größe erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

Die Konsumentengruppe ruft die Nachrichten als ein BLOB von Bytes ab, base64-codiert sie dann in eine einzelne JSON-Nutzlast und ruft dann Ihre Funktion auf. Wenn Ihre Funktion einen Fehler für eine der Nachrichten in einem Batch zurückgibt, wiederholt Lambda den gesamten Nachrichtenbatch, bis die Verarbeitung erfolgreich ist oder die Nachrichten ablaufen.

> **Note**
>
> Während Lambda-Ereignisfunktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB und Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten. Diese Einschränkung stellt sicher, dass die Ereignisquellenzuordnung Funktionsfehler und Wiederholungsversuche ordnungsgemäß verarbeiten kann.

Sie können die Parallelitätsnutzung einer bestimmten Funktion mithilfe der `ConcurrentExecutions` Metrik in Amazon CloudWatch überwachen. Weitere Hinweise zur Gleichzeitigkeitsskalierung finden Sie unter the section called “Konfigurieren reservierter Gleichzeitigkeit”.

Example Amazon-MQ-Datensatzereignisse

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
```

```
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/text-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
```

```
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      }
   ]
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
        "basicProperties": {
          "contentType": "text/plain",
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
```

```
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
          "replyTo": null,
          "expiration": "60000",
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
}
```

> **Note**
>
> Im RabbitMQ-Beispiel ist `pizzaQueue` der Name der RabbitMQ-Warteschlange und `/` der Name des virtuellen Hosts. Beim Empfang von Nachrichten listet die Ereignisquelle Nachrichten unter `pizzaQueue::/` auf.

Konfiguration der Amazon MQ-Ereignisquelle für Lambda

Themen

- Konfigurieren Sie die Netzwerksicherheit
- Die Zuordnung von Ereignisquellen erstellen

Konfigurieren Sie die Netzwerksicherheit

Um Lambda über die Zuordnung von Ereignisquellen Vollzugriff auf Amazon MQ zu gewähren, muss Ihr Broker entweder einen öffentlichen Endpunkt (öffentliche IP-Adresse) verwenden oder Sie müssen Zugriff auf die Amazon-VPC gewähren, in der Sie den Broker erstellt haben.

Wenn Sie Amazon MQ mit Lambda verwenden, erstellen Sie AWS PrivateLink VPC-Endpunkte, die Ihrer Event Function Zugriff auf die Ressourcen in Ihrer Amazon VPC gewähren.

> **Note**
>
> AWS PrivateLink VPC-Endpunkte sind für Funktionen mit Ereignisquellenzuordnungen erforderlich, die den Standardmodus (auf Abruf) für Ereignisabfragen verwenden. Wenn Ihre Ereignisquellenzuordnung den Bereitstellungsmodus verwendet, müssen Sie keine AWS PrivateLink VPC-Endpunkte konfigurieren.

Erstellen Sie einen Endpunkt, der den Zugriff auf die folgenden Ressourcen ermöglicht:

- Lambda – Erstellen Sie einen Endpunkt für den Lambda-Serviceprinzipal.
- AWS STS — Erstellen Sie einen Endpunkt für den, damit ein Service Principal eine Rolle AWS STS in Ihrem Namen übernehmen kann.
- Secrets Manager – Wenn Ihr Broker Secrets Manager zum Speichern von Anmeldeinformationen verwendet, erstellen Sie einen Endpunkt für Secrets Manager.

Alternativ konfigurieren Sie ein NAT-Gateway auf jedem öffentlichen Subnetz in der Amazon VPC. Weitere Informationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn Sie eine Zuordnung von Ereignisquellen für Amazon MQ erstellen, prüft Lambda, ob Elastic-Network-Schnittstelle (ENI) für die für Ihre Amazon VPC konfigurierten Subnetze und Sicherheitsgruppen bereits vorhanden sind. Findet Lambda vorhandene ENIs, versucht es, diese wiederzuverwenden. Andernfalls erstellt Lambda neue ENIs, um eine Verbindung mit der Ereignisquelle herzustellen und Ihre Funktion aufzurufen.

> **Note**
>
> Lambda-Ereignisfunktionen werden immer in VPCs ausgeführt, die dem Lambda-Service gehören. Die VPC-Konfiguration Ihrer Funktion hat keinen Einfluss auf die Zuordnung von Ereignisquellen. Nur die Netzwerkkonfiguration der Ereignisquelle bestimmt, wie Lambda sich mit Ihrer Ereignisquelle verbindet.

Konfigurieren Sie die Sicherheitsgruppen für die Amazon VPC, die Ihren Broker enthält. Standardmäßig verwendet Amazon MQ die folgenden Ports: 61617 (Amazon MQ für ActiveMQ) und 5671 (Amazon MQ für RabbitMQ).

- Eingehende Regeln – Erlauben Sie den gesamten Datenverkehr auf dem Standard-Broker-Port für die Sicherheitsgruppe, die mit Ihrer Ereignisquelle verbunden ist. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr — Erlauben Sie den gesamten Datenverkehr auf dem Port 443 für externe Ziele, wenn Ihre Funktion mit Diensten kommunizieren muss. AWS Alternativ können Sie auch eine selbstreferenzierende Sicherheitsgruppenregel verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen Diensten kommunizieren müssen. AWS
- Amazon VPC-Endpunkt-Eingangsregeln – Wenn Sie einen Amazon VPC-Endpunkt verwenden, muss die Sicherheitsgruppe, die mit Ihrem Amazon VPC-Endpunkt verbunden ist, eingehenden Verkehr auf Port 443 von der Broker-Sicherheitsgruppe zulassen.

Wenn Ihr Broker Authentifizierung verwendet, können Sie auch die Endpunktrichtlinie für den Secrets Manager-Endpunkt einschränken. Für den Aufruf der Secrets Manager-API verwendet Lambda Ihre Funktionsrolle und nicht den Lambda-Serviceprinzipal.

Example VPC-Endpunktrichtlinie – Secrets Manager-Endpunkt

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

Wenn Sie Amazon VPC-Endpunkte verwenden, AWS leitet Ihre API-Aufrufe zum Aufrufen Ihrer Funktion über das Elastic Network Interface (ENI) des Endpunkts weiter. Der Prinzipal des Lambda-Dienstes muss `lambda:InvokeFunction` für alle Rollen und Funktionen aufrufen, die diese ENI verwenden.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Um sicherzustellen, dass Ihre Ereignisquellenzuordnung Ihre Lambda-Ereignisfunktion aufrufen kann, muss die VPC-Endpunktrichtlinie dem Lambda-Serviceprinzipal erlauben, und aufzurufen. `sts:AssumeRole` `lambda:InvokeFunction` Wenn Sie Ihre VPC-Endpunktrichtlinien so einschränken, dass nur API-Aufrufe zugelassen werden, die aus Ihrem Unternehmen stammen, kann die Zuordnung von Ereignisquellen nicht richtig funktionieren, weshalb in diesen Richtlinien `"Resource": "*"` erforderlich ist.

Die folgenden Beispiel-VPC-Endpunktrichtlinien zeigen, wie der erforderliche Zugriff auf den Lambda-Serviceprinzipal für die AWS STS - und Lambda-Endpunkte gewährt wird.

Example VPC-Endpunktrichtlinie — AWS STS Endpunkt

```
{
      "Statement": [
          {
              "Action": "sts:AssumeRole",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
    }
```

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
```

```
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Die Zuordnung von Ereignisquellen erstellen

Erstellen Sie eine Ereignisquellenzuordnung, um Lambda anzuweisen, Datensätze von einem Amazon MQ-Broker an eine Lambda-Ereignisfunktion zu senden. Sie können mehrere Ereignisquellen-Zuweisungen erstellen, um gleiche Daten mit mehreren Funktionen oder Elemente aus mehreren Streams mit nur einer Funktion zu verarbeiten.

Um Ihre Funktion für das Lesen von Amazon MQ zu konfigurieren, fügen Sie die erforderlichen Berechtigungen hinzu und erstellen einen MQ-Auslöser in der Lambda-Konsole.

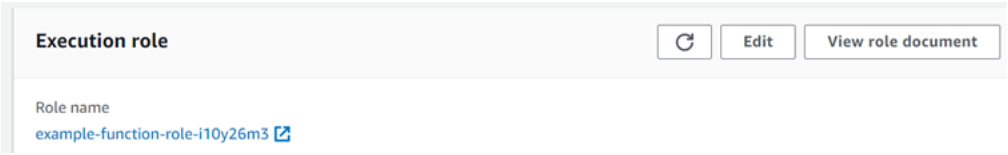
Um Datensätze von einem Amazon MQ-Broker zu lesen, benötigt Ihre Lambda Event Function die folgenden Berechtigungen. Sie gewähren Lambda die Erlaubnis, mit Ihrem Amazon MQ-Broker und den zugrunde liegenden Ressourcen zu interagieren, indem Sie Ihrer Funktionsausführungsrolle Berechtigungsanweisungen hinzufügen:

- mq: DescribeBroker
- Verwalter von Geheimnissen: GetSecretValue
- ec2: CreateNetworkInterface
- ec2: DeleteNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeSecurityGroups
- ec2: DescribeSubnets
- ec2: DescribeVpcs
- Logs: CreateLogGroup
- Logs: CreateLogStream
- Logs: PutLogEvents

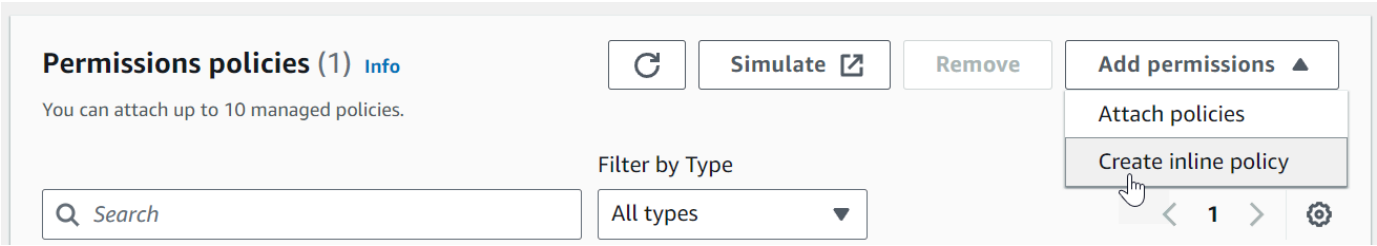
> **Note**
>
> Wenn Sie einen verschlüsselten, von Kunden verwalteten Schlüssel verwenden, fügen Sie auch die `kms:Decrypt`-Berechtigung hinzu.

So fügen Sie Berechtigungen hinzu und erstellen einen Auslöser

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



5. Wählen Sie Add permissions (Berechtigungen hinzufügen) und dann Create inline policy (Inline-Richtlinie erstellen) aus.



6. Wählen Sie im Abschnitt Richtlinien-Editor JSON aus. Geben Sie die folgende Richtlinie ein: Ihre Funktion benötigt diese Berechtigungen, um von einem Amazon MQ-Broker zu lesen.

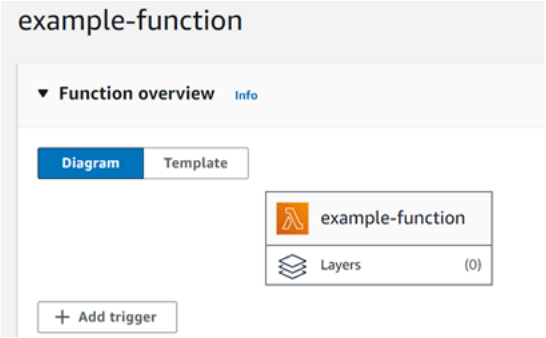
   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "mq:DescribeBroker",
          "secretsmanager:GetSecretValue",
```

```
            "ec2:CreateNetworkInterface",
            "ec2:DeleteNetworkInterface",
            "ec2:DescribeNetworkInterfaces",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeSubnets",
            "ec2:DescribeVpcs",
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
        ],
        "Resource": "*"
    }
  ]
}
```

> **Note**
>
> Wenn Sie einen verschlüsselten, vom Kunden verwalteten Schlüssel verwenden, müssen Sie auch die `kms:Decrypt`-Berechtigung hinzufügen.

7. Wählen Sie Weiter aus. Geben Sie einen Richtliniennamen ein und wählen Sie anschließend Create policy (Richtlinie erstellen) aus.
8. Gehen Sie zurück zu Ihrer Funktion in der Lambda-Konsole. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).



9. Wählen Sie den MQ-Auslösertyp.
10. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Optionen für Amazon-MQ-Ereignisquellen.

- MQ-Broker – Wählen Sie einen Amazon-MQ-Broker aus.

- Batchgröße – Legen Sie die maximale Anzahl von Nachrichten fest, die in einem einzelnen Batch abgerufen werden sollen.
- Name der Warteschlange – Geben Sie die zu konsumierende Amazon-MQ-Warteschlange ein.
- Konfiguration des Zugriffs – Geben Sie die Informationen zum virtuellen Host und das Secrets Manager-Geheimnis ein, in dem Ihre Broker-Anmeldeinformationen gespeichert sind.
- Auslöser aktivieren – Deaktivieren Sie den Auslöser, um die Verarbeitung von Datensätzen anzuhalten.

Um den Auslöser zu aktivieren oder zu deaktivieren (oder zu löschen), wählen Sie den MQ-Auslöser im Designer aus. Verwenden Sie zum Neukonfigurieren des Auslösers die API-Vorgänge für die Ereignisquellen-Zuweisung.

Parameter für die Zuordnung von Ereignisquellen

Alle Lambda-Ereignisquellentypen verwenden dieselben API-Operationen CreateEventSourceMapping und UpdateEventSourceMapping. Allerdings gelten nur einige der Parameter für Amazon MQ und RabbitMQ.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
Enabled	N	true	Keine
FunctionName	Y	N/A	Keine
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet
MaximumBatchingWindowInSeconds	N	500 ms	Batching-Verhalten
Warteschlangen	N	N/A	Der Name der zu verwendenden Zielwarteschlange des Amazon-MQ-Brokers.

Parameter	Erforderlich	Standard	Hinweise
SourceAccessConfigurations	N	N/A	Für ActiveMQ BASIC_AUTH-Anmeldeinformationen. Für RabbitMQ können sowohl BASIC_AUTH Anmeldeinformationen als auch VIRTUAL_HOST Informationen enthalten sein.

Ereignisse aus einer Amazon MQ-Ereignisquelle filtern

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Event-Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für Amazon MQ-Ereignisquellen.

> **Note**
>
> Amazon-MQ-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `data`-Schlüssel.

Themen

- Grundlagen der Amazon MQ-Ereignisfilterung

Grundlagen der Amazon MQ-Ereignisfilterung

Angenommen, Ihre Amazon-MQ-Nachrichtenwarteschlange enthält Nachrichten entweder im gültigen JSON-Format oder als einfache Zeichenfolgen. Ein Beispieldatensatz würde wie folgt aussehen, wobei die Daten im `data`-Feld in eine Base64-kodierte Zeichenfolge umgewandelt wurden.

ActiveMQ

```
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  49
                ]
```

```
            },
            "header2": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

Sowohl für Active MQ- als auch für Rabbit MQ-Broker können Sie die Ereignisfilterung verwenden, um Datensätze anhand des `data`-Schlüssels zu filtern. Angenommen, Ihre Amazon-MQ-Warteschlange enthält Nachrichten im folgenden JSON-Format.

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

Um nur die Datensätze zu filtern, bei denen das `timeout`-Feld größer als 0 ist, würde das `FilterCriteria`-Objekt wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer AWS SAM Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

Mit Amazon MQ können Sie auch Datensätze filtern, bei denen die Nachricht eine einfache Zeichenfolge ist. Angenommen, Sie möchten nur Datensätze verarbeiten, bei denen die Meldung mit „Ergebnis:“ beginnt. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "data": [
        {
        "prefix": "Result: "
```

```
        }
    ]
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer Vorlage hinzufügen. AWS SAM

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
```

```
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ MQ-Nachrichten müssen UTF-8 codierte Zeichenfolgen sein, entweder einfache Zeichenfolgen oder im JSON-Format. Das liegt daran, dass Lambda Amazon MQ-Byte-Arrays dekodiert, UTF-8 bevor Filterkriterien angewendet werden. Wenn Ihre Nachrichten eine andere Kodierung verwenden, z. UTF-16 B. ASCII, oder wenn das Nachrichtenformat nicht dem `FilterCriteria` Format entspricht, verarbeitet Lambda nur Metadatenfilter. In der folgenden Tabelle ist das Verhalten zusammengefasst:

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
Einfache Zeichenfolge	Einfache Zeichenfolge	Lambda filtert basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Gültiges JSON	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Einfache Zeichenfolge	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
Non-UTF-8 codierte Zeichenfolge	JSON, einfache Zeichenfolge oder kein Muster	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.

Beheben Sie Fehler Amazon MQ-Zuordnung von Ereignisquellen

Wenn eine Lambda-Ereignisfunktion auf einen nicht behebbaren Fehler stößt, stoppt Ihr Amazon MQ MQ-Kunde die Verarbeitung von Datensätzen. Alle anderen Konsumenten können die Verarbeitung fortsetzen, sofern sie nicht auf denselben Fehler stoßen. Um die potenzielle Ursache für einen gestoppten Konsumenten zu ermitteln, überprüfen Sie das `StateTransitionReason`-Feld in den Rücksendedetails Ihres `EventSourceMapping` auf einen der folgenden Codes:

`ESM_CONFIG_NOT_VALID`

Die Konfiguration der Ereignisquellen-Zuweisung ist ungültig.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda konnte die Ereignisquelle nicht authentifizieren.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda verfügt nicht über die erforderlichen Berechtigungen für den Zugriff auf die Ereignisquelle.

`FUNCTION_CONFIG_NOT_VALID`

Die Konfiguration der Funktion ist ungültig.

Datensätze bleiben auch unbearbeitet, wenn Lambda sie aufgrund ihrer Größe fallen lässt. Die Größenbeschränkung für Lambda-Datensätze beträgt 6 MB. Um Nachrichten bei Funktionsfehlern erneut zuzustellen, können Sie eine Warteschlange für unzustellbare Nachrichten (DLQ) verwenden. Weitere Informationen finden Sie unter Erneute Zustellung von Nachrichten und DLQ-Handhabung auf der Apache-ActiveMQ-Website und Zuverlässigkeits-Leitfaden auf der RabbitMQ-Website.

> **Note**
>
> Lambda unterstützt keine benutzerdefinierten Richtlinien für die erneute Bereitstellung. Stattdessen verwendet Lambda eine Richtlinie mit den Standardwerten von der Seite Richtlinie für erneute Zustellung auf der Apache-ActiveMQ-Website, wobei `maximumRedeliveries` auf 6 gesetzt ist.

Verwenden AWS Lambda mit Amazon RDS

Sie können eine Lambda-Ereignisfunktion direkt und über einen Amazon RDS-Proxy mit einer Amazon Relational Database Service (Amazon RDS) -Datenbank verbinden. Direkte Verbindungen sind in einfachen Szenarien nützlich, und Proxys werden für die Produktion empfohlen. Ein Datenbank-Proxy verwaltet einen Pool gemeinsam genutzter Datenbankverbindungen, sodass Ihre Funktion eine hohe Gleichkeitigkeitsstufe erreichen kann, ohne dass die Datenbankverbindungen erschöpft werden.

Informationen zur Entscheidung, ob Sie Amazon RDS oder DynamoDB für Ihren Workload verwenden sollten, finden Sie unter. the section called “Amazon RDS im Vergleich zu DynamoDB”

Wir empfehlen die Verwendung von Amazon RDS Proxy for Lambda Event Functions, die häufig kurze Datenbankverbindungen herstellen oder eine große Anzahl von Datenbankverbindungen öffnen und schließen. Weitere Informationen finden Sie unter Automatisches Verbinden einer Lambda-Funktion und einer DB-Instance im Amazon Relational Database Service Developer Guide.

> **Tip**
>
> Um eine Lambda-Ereignisfunktion schnell mit einer Amazon RDS-Datenbank zu verbinden, können Sie den in der Konsole integrierten Assistenten verwenden. Gehen Sie wie folgt vor, um den Assistenten zu öffnen:
>
> 1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
> 2. Wählen Sie die Funktion aus, mit der Sie eine Datenbank verbinden möchten.
> 3. Wählen Sie auf der Registerkarte Konfiguration die Option RDS-Datenbanken aus.
> 4. Wählen Sie Verbindung zur RDS-Datenbank herstellen aus.
>
> Nachdem Sie Ihre Funktion mit einer Datenbank verbunden haben, können Sie über die Option Proxy hinzufügen einen Proxy erstellen.

Konfiguration Ihrer Event-Funktion für die Verwendung mit RDS-Ressourcen

In der Lambda-Konsole können Sie Amazon-RDS-Datenbankinstanzen und Proxy-Ressourcen bereitstellen und konfigurieren. Sie können dies tun, indem Sie auf der Registerkarte Konfiguration zu den RDS-Datenbanken navigieren. Alternativ können Sie auch Verbindungen zu Lambda Event

Functions in der Amazon RDS-Konsole erstellen und konfigurieren. Wenn Sie eine RDS-Datenbank-Instance für die Verwendung mit Lambda konfigurieren, beachten Sie die folgenden Kriterien:

- Um eine Verbindung zu einer Datenbank herzustellen, muss sich die Funktion in derselben Amazon VPC befinden, in der Ihre Datenbank ausgeführt wird.
- Sie können Amazon-RDS-Datenbanken mit MySQL-, MariaDB-, PostgreSQL- oder Microsoft-SQL-Server-Engines verwenden.
- Sie können Aurora-DB-Cluster auch mit MySQL- oder PostgreSQL-Engines verwenden.
- Sie müssen ein Secrets-Manager-Geheimnis für die Authentifizierung der Datenbank angeben.
- Eine IAM-Rolle muss die Berechtigung zur Verwendung des Geheimnisses sowie eine Vertrauensrichtlinie bereitstellen, mit der Amazon RDS die Rolle übernehmen kann.
- Der IAM-Prinzipal, der die Konsole verwendet, um die Amazon-RDS-Ressource zu konfigurieren und sie mit Ihrer Funktion zu verbinden, muss über die folgenden Berechtigungen verfügen:

Beispiel für eine Berechtigungsrichtlinie

> **Note**
>
> Sie benötigen die Amazon-RDS-Proxy Berechtigungen nur, wenn Sie einen Amazon-RDS-Proxy konfigurieren, um einen Pool Ihrer Datenbankverbindungen zu verwalten.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
```

```
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
```

```
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS berechnet einen Stundensatz für Proxys, der auf der Größe der Datenbank-Instance basiert. Weitere Informationen finden Sie unter RDS-Proxy-Preise. Weitere Informationen zu Proxy-Verbindungen finden Sie unter Verwendung von Amazon-RDS-Proxy im Amazon-RDS-Benutzerhandbuch.

SSL/TLS Anforderungen für Amazon RDS-Verbindungen

Um sichere SSL/TLS Verbindungen zu einer Amazon RDS-Datenbank-Instance herzustellen, muss Ihre Lambda Event Function die Identität des Datenbankservers mithilfe eines vertrauenswürdigen Zertifikats verifizieren. Lambda verarbeitet diese Zertifikate je nach Art Ihres Bereitstellungspakets unterschiedlich:

- ZIP-Dateiarchive: Die Zertifikatsverarbeitung variiert je nach Laufzeit:
  - Node.js 18 und früher: Lambda schließt automatisch CA-Zertifikate und RDS-Zertifikate ein.
  - Node.js 20 und später: Lambda lädt standardmäßig keine zusätzlichen CA-Zertifikate mehr. Legen Sie die Umgebungsvariable `NODE_EXTRA_CA_CERTS` auf `/var/runtime/ca-cert.pem` fest.

  Es kann bis zu 4 Wochen dauern, bis Amazon RDS-Zertifikate AWS-Regionen zu den verwalteten Lambda-Laufzeiten hinzugefügt werden.
- Container-Images: AWS Basis-Images enthalten nur CA-Zertifikate. Wenn Ihre Funktion eine Verbindung zu einer Amazon-RDS-Datenbank-Instance herstellt, müssen Sie die entsprechenden Zertifikate in Ihr Container-Image einbinden. Laden Sie in Ihrem Dockerfile das Zertifikatspaket herunter, das dem entspricht, AWS-Region wo Sie Ihre Datenbank hosten. Beispiel:

```
RUN curl https://truststore.pki.rds.amazonaws.com/us-east-1/us-east-1-bundle.pem -o /
us-east-1-bundle.pem
```

Mit diesem Befehl wird das Amazon-RDS-Zertifikatspaket heruntergeladen und im absoluten Pfad `/us-east-1-bundle.pem` im Stammverzeichnis Ihres Containers gespeichert. Beim Konfigurieren der Datenbankverbindung in Ihrem Funktionscode müssen Sie genau diesen Pfad angeben. Beispiel:

Node.js

Die `readFileSync` Funktion ist erforderlich, da Node.js Datenbankclients den tatsächlichen Zertifikatsinhalt im Speicher benötigen, nicht nur den Pfad zur Zertifikatsdatei. Ohne „`readFileSync`“ interpretiert der Client die Pfadzeichenfolge als Zertifikatsinhalt, was zu dem Fehler „Selbstsigniertes Zertifikat in der Zertifikatskette“ führt.

Example Node.js Verbindungskonfiguration für die OCI-Funktion

```
import { readFileSync } from 'fs';

// ...

let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: {
        ca: readFileSync('/us-east-1-bundle.pem') // Load RDS certificate content
 from file into memory
    }
};
```

Python

Example Python-Verbindungskonfiguration für die OCI-Funktion

```
connection = pymysql.connect(
    host=proxy_host_name,
    user=db_username,
    password=token,
```

```
    db=db_name,
    port=port,
    ssl={'ca': '/us-east-1-bundle.pem'}  #Path to the certificate in container
)
```

Java

Für Java-Funktionen mit JDBC-Verbindungen muss die Verbindungszeichenfolge Folgendes enthalten:

- `useSSL=true`
- `requireSSL=true`
- Einen `sslCA`-Parameter, der auf den Speicherort des Amazon-RDS-Zertifikats im Container-Image verweist

Example Java-Verbindungszeichenfolge für die OCI-Funktion

```
// Define connection string
String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true&sslCA=/us-east-1-bundle.pem", // Path to the certificate
 in container
        System.getenv("ProxyHostName"),
        System.getenv("Port"),
        System.getenv("DBName"));
```

.NET

Example.NET-Verbindungszeichenfolge für MySQL-Verbindung in der OCI-Funktion

```
/// Build the Connection String with the Token
string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +
                         $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                         $"Pwd={authToken};" +
                         "SslMode=Required;" +
                         "SslCa=/us-east-1-bundle.pem";  // Path to the certificate
 in container
```

Go

Für Go-Funktionen mit MySQL-Verbindungen laden Sie das Amazon-RDS-Zertifikat in einen Zertifikatspool und registrieren es beim MySQL-Treiber. Die Verbindungszeichenfolge muss dann mithilfe des `tls`-Parameters auf diese Konfiguration verweisen.

Example Go-Code für MySQL-Verbindung in der OCI-Funktion

```
import (
    "crypto/tls"
    "crypto/x509"
    "os"
    "github.com/go-sql-driver/mysql"
)

...

// Create certificate pool and register TLS config
rootCertPool := x509.NewCertPool()
pem, err := os.ReadFile("/us-east-1-bundle.pem")  // Path to the certificate in
 container
if err != nil {
    panic("failed to read certificate file: " + err.Error())
}
if ok := rootCertPool.AppendCertsFromPEM(pem); !ok {
    panic("failed to append PEM")
}

mysql.RegisterTLSConfig("custom", &tls.Config{
    RootCAs: rootCertPool,
})

dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?allowCleartextPasswords=true&tls=custom",
    dbUser, authenticationToken, dbEndpoint, dbName,
)
```

Ruby

Example Ruby-Verbindungskonfiguration für die OCI-Funktion

```
conn = Mysql2::Client.new(
    host: endpoint,
    username: user,
```

```
    password: token,
    port: port,
    database: db_name,
    sslca: '/us-east-1-bundle.pem',  # Path to the certificate in container
    sslverify: true
)
```

In einer Lambda-Event-Funktion eine Verbindung zu einer Amazon RDS-Datenbank herstellen

Die folgenden Codebeispiele zeigen, wie eine Lambda-Ereignisfunktion implementiert wird, die eine Verbindung zu einer Amazon RDS-Datenbank herstellt. Die Funktion stellt eine einfache Datenbankanfrage und gibt das Ergebnis zurück.

> **Note**
>
> Diese Codebeispiele gelten nur für ZIP-Bereitstellungspakete. Wenn Sie Ihre Funktion mithilfe eines Container-Images bereitstellen, müssen Sie die Amazon-RDS-Zertifikatsdatei in Ihrem Funktionscode angeben, wie im vorherigen Abschnitt beschrieben.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion mithilfe von .NET.

```
using System.Data;
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
        string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

 $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +
```

```
$"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


       try
       {
           await using var connection = new MySqlConnection(connectionString);
           await connection.OpenAsync();

           const string sql = "SELECT @param1 + @param2 AS Sum";

           await using var command = new MySqlCommand(sql, connection);
           command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
"0"));
           command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
"0"));

           await using var reader = await command.ExecuteReaderAsync();
           if (await reader.ReadAsync())
           {
               int result = reader.GetInt32("Sum");

               //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
sum is: 45\"}","isBase64Encoded":false}
               return new APIGatewayProxyResponse
               {
                   StatusCode = 200,
                   Body = JsonSerializer.Serialize(new { message = $"The sum is:
{result}" })
               };
           }

       }
       catch (Exception ex)
       {
           Console.WriteLine($"Error: {ex.Message}");
       }

       return new APIGatewayProxyResponse
       {
           StatusCode = 500,
           Body = JsonSerializer.Serialize(new { error = "Internal server
error" })
```

```
        };
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Go.

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}
```

```
func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }
```

```
 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }
```

```
        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mithilfe von JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
```

```
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mithilfe von TypeScript.

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });
```

```
    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit PHP.

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
```

```
            'region' => getenv('AWS_REGION'),
        ];

        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
```

```
    }

    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Python.

```
import json
import os
import boto3
import pymysql
```

```
# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Ruby.

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )
```

```
  begin
    conn = Mysql2::Client.new(
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Rust.

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
```

```
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
```

```
        db_user = db_username
    );

    let signable_request =
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
```

```
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

Verarbeitung von Ereignisbenachrichtigungen von Amazon RDS

Sie können Lambda verwenden, um Ereignisbenachrichtigungen aus einer Amazon-RDS-Datenbank zu verarbeiten. Amazon RDS sendet Benachrichtigungen an ein Amazon Simple Notification Service (Amazon SNS) -Thema, das Sie so konfigurieren können, dass es eine Lambda-Ereignisfunktion aufruft. Amazon SNS wickelt die Nachricht von Amazon RDS in ein eigenes Ereignisdokument und sendet sie an Ihre Funktion.

Weitere Informationen zum Konfigurieren einer Amazon-RDS-Datenbank zum Senden von Benachrichtigungen finden Sie unter Verwendung von Amazon-RDS-Ereignisbenachrichtigungen.

Example Amazon-RDS-Nachricht in einem Amazon-SNS-Ereignis

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
```

```
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
              "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
              "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
              "MessageAttributes": {},
              "Type": "Notification",
              "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
              "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
              "Subject": "RDS Notification Message"
            }
          }
        ]
      }
```

Vollständiges Lambda- und Amazon-RDS-Tutorial

- Verwenden einer Lambda-Funktion für den Zugriff auf eine Amazon RDS-Datenbank — Im Amazon RDS-Benutzerhandbuch erfahren Sie, wie Sie eine Lambda-Ereignisfunktion verwenden, um Daten über einen Amazon RDS-Proxy in eine Amazon RDS-Datenbank zu schreiben. Ihre Lambda-Ereignisfunktion liest Datensätze aus einer Amazon SQS SQS-Warteschlange und schreibt jedes Mal, wenn eine Nachricht hinzugefügt wird, neue Elemente in eine Tabelle in Ihrer Datenbank.

Wählen Sie einen Datenbankdienst für Ihre Lambda-based Anwendungen

Viele serverlose Anwendungen müssen Daten speichern und abrufen. AWS bietet mehrere Datenbankoptionen, die mit Lambda Event Functions funktionieren. Zwei der beliebtesten Optionen sind Amazon DynamoDB, ein NoSQL-Datenbankservice, und Amazon RDS, eine traditionelle relationale Datenbanklösung. In den folgenden Abschnitten werden die wichtigsten Unterschiede

zwischen diesen Services bei der Verwendung mit Lambda erläutert und Sie bei der Auswahl des passenden Datenbankservices für Ihre Serverless-Anwendung unterstützt.

Weitere Informationen über die anderen von AWS angebotenen Datenbankdienste und ein allgemeines Verständnis ihrer Anwendungsfälle und Kompromisse finden Sie unter Auswahl eines AWS Datenbankdienstes. Alle AWS Datenbankdienste sind mit Lambda kompatibel, aber möglicherweise sind nicht alle für Ihren speziellen Anwendungsfall geeignet.

Welche Möglichkeiten haben Sie bei der Auswahl eines Datenbankservices für Lambda?

AWS bietet mehrere Datenbankdienste. Für Serverless-Anwendungen sind DynamoDB und Amazon RDS zwei der beliebtesten Optionen.

- DynamoDB ist ein vollständig verwalteter NoSQL-Datenbankservice, der für Serverless-Anwendungen optimiert ist. Er bietet nahtlose Skalierung und eine konsistente Leistung im einstelligen Millisekundenbereich in jeder Größenordnung.
- Amazon RDS ist ein verwalteter relationaler Datenbankservice, der mehrere Datenbank-Engines unterstützt, darunter MySQL und PostgreSQL. Er bietet vertraute SQL-Funktionen mit verwalteter Infrastruktur.

Empfehlungen, falls Sie Ihre Anforderungen bereits kennen

Wenn Sie sich über Ihre Anforderungen bereits im Klaren sind, finden Sie hier unsere grundlegenden Empfehlungen:

Wir empfehlen DynamoDB für Serverless-Anwendungen, die eine konsistente Leistung mit niedriger Latenz und automatische Skalierung benötigen und keine komplexen Verknüpfungen oder Transaktionen erfordern. Aufgrund seines serverlosen Charakters eignet es sich besonders gut für Lambda-based Anwendungen.

Amazon RDS ist die bessere Wahl, wenn Sie komplexe SQL-Abfragen oder Verknüpfungen benötigen oder bereits Anwendungen mit relationalen Datenbanken verwenden. Beachten Sie jedoch, dass die Verbindung von Lambda Event Functions mit Amazon RDS eine zusätzliche Konfiguration erfordert und sich auf die Kaltstartzeiten auswirken kann.

Was Sie bei der Auswahl eines Datenbankservices beachten sollten

Bei der Auswahl zwischen DynamoDB und Amazon RDS für Ihre Lambda-Anwendungen sollten Sie die folgenden Faktoren berücksichtigen:

- Verbindungsmanagement und Kaltstarts
- Datenzugriffsmuster
- Abfragekomplexität
- Anforderungen für Datenkonsistenz.
- Skalierungsmerkmale
- Kostenmodell

Wenn Sie diese Faktoren verstehen, können Sie die Option auswählen, die Ihren spezifischen Anwendungsfällen am besten entspricht.

Verbindungsmanagement und Kaltstarts

- DynamoDB verwendet eine HTTP-API für alle Operationen. Lambda-Ereignisfunktionen können sofortige Anfragen stellen, ohne Verbindungen aufrechtzuerhalten, was zu einer besseren Kaltstartleistung führt. Jede Anfrage wird mithilfe von AWS Anmeldeinformationen ohne Verbindungsaufwand authentifiziert.
- Amazon RDS erfordert die Verwaltung von Verbindungspools, da es herkömmliche Datenbankverbindungen verwendet. Dies kann sich auf Kaltstarts auswirken, da neue Lambda-Instances Verbindungen herstellen müssen. Sie müssen Strategien für das Verbindungspooling implementieren und möglicherweise den Amazon-RDS-Proxy verwenden, um Verbindungen effektiv zu verwalten. Beachten Sie, dass für die Verwendung des Amazon-RDS-Proxy zusätzliche Kosten anfallen.

Datenzugriffsmuster

- DynamoDB eignet sich am besten für bekannte Zugriffsmuster und Einzeltabellendesigns. Es ist ideal für Lambda-Anwendungen, die einen konsistenten Zugriff mit geringer Latenz auf Daten basierend auf Primärschlüsseln oder Sekundärindizes benötigen.
- Amazon RDS bietet Flexibilität für komplexe Abfragen und sich ändernde Zugriffsmuster. Es ist besser geeignet, wenn Ihre Lambda-Ereignisfunktionen einzigartige, maßgeschneiderte Abfragen oder komplexe Verknüpfungen über mehrere Tabellen hinweg ausführen müssen.

Abfragekomplexität

- DynamoDB zeichnet sich durch einfache, schlüsselbasierte Operationen und vordefinierte Zugriffsmuster aus. Komplexe Abfragen müssen um Indexstrukturen herum entworfen werden, und Verknüpfungen müssen im Anwendungscode behandelt werden.
- Amazon RDS unterstützt komplexe SQL-Abfragen mit Verknüpfungen, Unterabfragen und Aggregationen. Dies kann Ihren Lambda Event Function-Code vereinfachen, wenn komplexe Datenoperationen erforderlich sind.

Anforderungen für Datenkonsistenz.

- DynamoDB bietet sowohl Optionen für letztendliche als auch für starke Konsistenz, wobei starke Konsistenz für Lesevorgänge einzelner Elemente verfügbar ist. Transaktionen werden unterstützt, jedoch mit einigen Einschränkungen.
- Amazon RDS bietet vollständige ACID-Konformität (Atomizität, Konsistenz, Isolierung und Zuverlässigkeit) sowie Unterstützung für komplexe Transaktionen. Wenn Ihre Lambda-Ereignisfunktionen komplexe Transaktionen oder eine hohe Konsistenz über mehrere Datensätze hinweg erfordern, ist Amazon RDS möglicherweise besser geeignet.

Skalierungsmerkmale

- DynamoDB skaliert automatisch mit Ihrer Workload. Es kann plötzliche Verkehrsspitzen von Lambda Event Functions ohne vorherige Bereitstellung bewältigen. Sie können den On-Demand-Kapazitätsmodus verwenden, um nur für das zu bezahlen, was Sie tatsächlich nutzen. Dies entspricht perfekt dem Skalierungsmodell von Lambda.
- Amazon RDS verfügt über eine feste Kapazität, die auf der von Ihnen ausgewählten Instance-Größe basiert. Wenn mehrere Lambda-Ereignisfunktionen versuchen, gleichzeitig eine Verbindung herzustellen, überschreiten Sie möglicherweise Ihr Verbindungskontingent. Sie müssen die Verbindungspools sorgfältig verwalten und möglicherweise eine Wiederholungslogik implementieren.

Kostenmodell

- Die Preisgestaltung von DynamoDB ist gut auf Serverless-Anwendungen abgestimmt. Mit On-Demand-Kapazität zahlen Sie nur für die tatsächlichen Lese- und Schreibvorgänge, die von Ihren Lambda Event Functions ausgeführt werden. Für Leerlaufzeiten fallen keine Gebühren an.

- Amazon RDS berechnet Gebühren für die laufende Instance unabhängig von der Nutzung. Dies kann bei sporadischen Workloads, wie sie in Serverless-Anwendungen typisch sind, weniger kosteneffizient sein. Für Workloads mit hohem Durchsatz und konstanter Nutzung kann es jedoch wirtschaftlicher sein.

Erste Schritte mit dem von Ihnen ausgewählten Datenbankservice

Nachdem Sie die Auswahlkriterien für DynamoDB und Amazon RDS sowie die wichtigsten Unterschiede kennengelernt haben, können Sie die Option wählen, die Ihren Bedürfnissen am besten entspricht, und mithilfe der folgenden Ressourcen loslegen.

DynamoDB

Nutzen Sie die folgenden Ressourcen, um mit DynamoDB zu beginnen

- Eine Einführung in den DynamoDB-Service finden Sie im Entwicklerhandbuch für Amazon DynamoDB unter Was ist DynamoDB?.
- Folgen Sie dem Tutorial „Lambda mit API Gateway verwenden“, um ein Beispiel für die Verwendung einer Lambda-Ereignisfunktion zur Ausführung von CRUD-Vorgängen in einer DynamoDB-Tabelle als Antwort auf eine API-Anfrage zu sehen.
- Lesen Sie Programming with DynamoDB and the AWS SDKs im Amazon DynamoDB Developer Guide, um mehr darüber zu erfahren, wie Sie mithilfe eines der SDKs von Ihrer Lambda Event Function aus auf DynamoDB zugreifen können. AWS

Amazon RDS

Nutzen Sie die folgenden Ressourcen, um mit Amazon RDS zu beginnen

- Eine Einführung in den Amazon-RDS-Service finden Sie im Abschnitt Was ist Amazon Relational Database Service (Amazon RDS)? im Benutzerhandbuch zu Amazon Relational Database Service.
- Folgen Sie dem Tutorial Verwenden einer Lambda-Funktion für den Zugriff auf eine Amazon-RDS-Datenbank im Benutzerhandbuch zu Amazon Relational Database Service.
- Weitere Informationen zur Verwendung von Lambda mit Amazon RDS finden Sie unter the section called “RDS”.

Verarbeitung von Amazon-S3-Ereignisbenachrichtigungen mit Lambda

Sie können Lambda verwenden, um Ereignisbenachrichtigungen von Amazon Simple Storage Service zu verarbeiten. Amazon S3 kann ein Ereignis an eine Lambda-Ereignisfunktion senden, wenn ein Objekt erstellt oder gelöscht wird. Sie konfigurieren Benachrichtigungseinstellungen für einen Bucket und erteilen die Amazon-S3-Berechtigung zum Aufrufen einer Funktion in der ressourcenbasierten Berechtigungsrichtlinie der Funktion.

> **Warning**
>
> Wenn Ihre Lambda-Ereignisfunktion denselben Bucket verwendet, der sie auslöst, kann dies dazu führen, dass die Funktion in einer Schleife ausgeführt wird. Wenn der Bucket z. B. eine Funktion immer dann auslöst, wenn ein Objekt hochgeladen wird, und die Funktion ein Objekt in den Bucket hochlädt, löst die Funktion sich indirekt selbst aus. Um dies zu vermeiden, verwenden Sie zwei Buckets oder konfigurieren Sie den Auslöser so, dass er nur für einen Präfix gilt, der für eingehende Objekte verwendet wird.

Amazon S3 ruft Ihre Funktion asynchron mit einem Ereignis auf, das Details über das Objekt enthält. Das folgende Beispiel zeigt ein Ereignis, das Amazon S3 gesendet hat, als ein Bereitstellungspaket in Amazon S3 hochgeladen wurde.

Example Amazon-S3-Benachrichtigungsereignis

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
```

```
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

Um Ihre Funktion aufzurufen, benötigt Amazon S3 die Berechtigung von der ressourcenbasierten Richtlinie der Funktion. Wenn Sie einen Amazon-S3-Auslöser in der Lambda-Konsole konfigurieren, ändert die Konsole die ressourcenbasierte Richtlinie so, dass Amazon S3 die Funktion aufrufen kann, wenn der Bucket-Name und die Konto-ID übereinstimmen. Wenn Sie die Benachrichtigung in Amazon S3 konfigurieren, verwenden Sie die Lambda-API, um die Richtlinie zu aktualisieren. Sie können auch die Lambda-API verwenden, um einem anderen Konto Berechtigungen zu erteilen oder die Berechtigung auf einen bestimmten Alias zu beschränken.

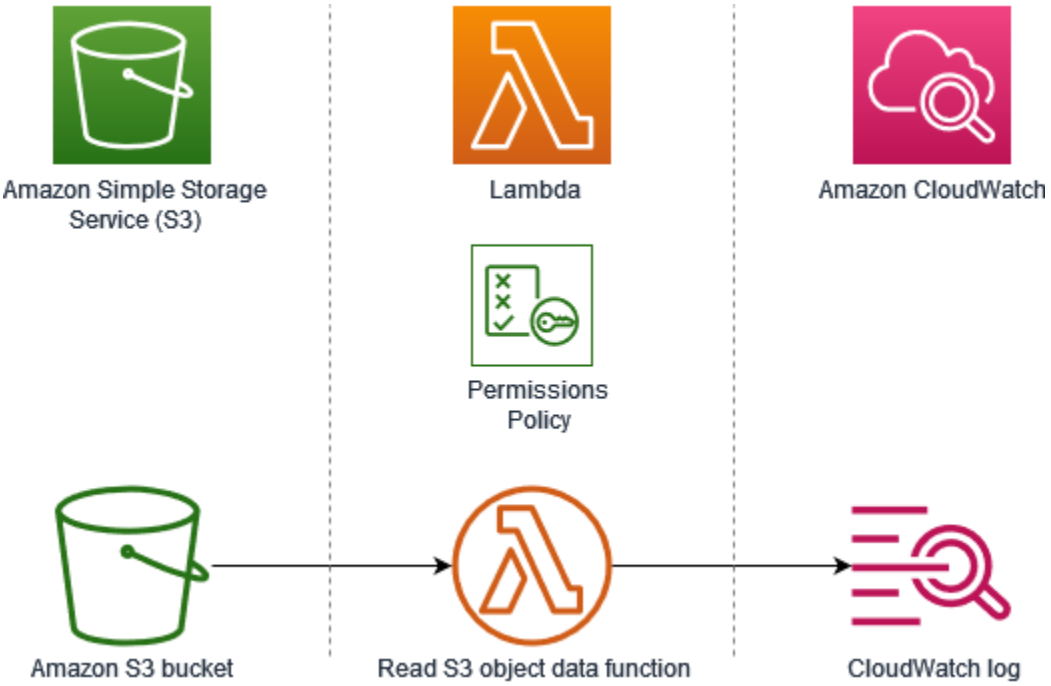
Wenn Ihre Funktion das AWS SDK zur Verwaltung von Amazon S3 S3-Ressourcen verwendet, benötigt sie in ihrer Ausführungsrolle auch Amazon S3 S3-Berechtigungen.

Themen

- Tutorial: Verwenden eines Amazon S3 S3-Triggers zum Aufrufen einer Lambda-Ereignisfunktion
- Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern

Tutorial: Verwenden eines Amazon S3 S3-Triggers zum Aufrufen einer Lambda-Ereignisfunktion

In diesem Tutorial verwenden Sie die Konsole, um eine Lambda-Ereignisfunktion zu erstellen und einen Trigger für einen Amazon Simple Storage Service (Amazon S3) -Bucket zu konfigurieren. Jedes Mal, wenn Sie Ihrem Amazon S3 S3-Bucket ein Objekt hinzufügen, wird Ihre Funktion ausgeführt und der Objekttyp wird in Amazon CloudWatch Logs ausgegeben.

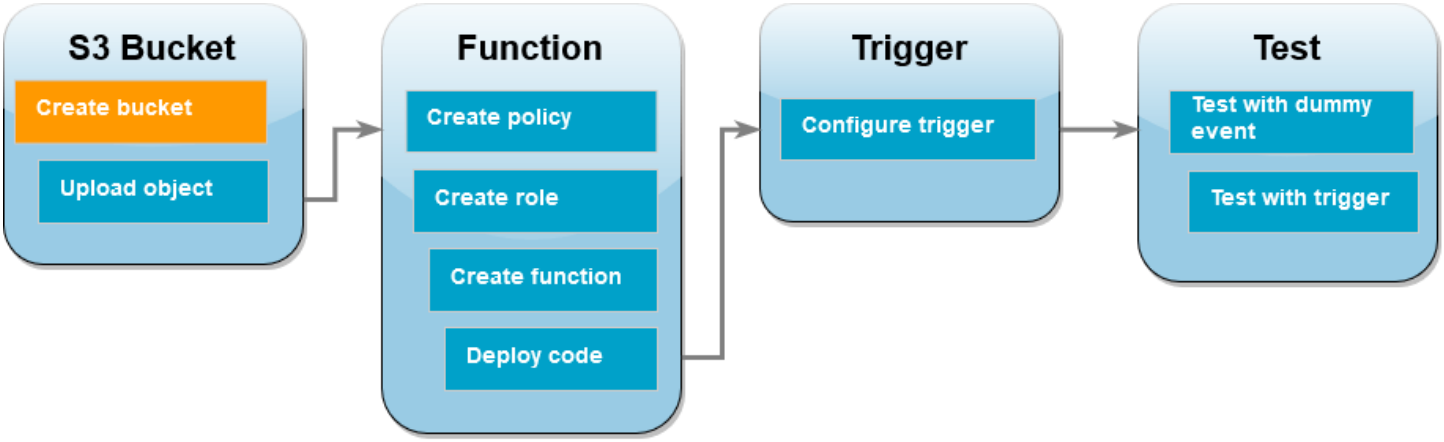


Dieses Tutorial zeigt, wie man:

1. Erstellen Sie einen Amazon-S3-Bucket.
2. Erstellen Sie eine Lambda-Ereignisfunktion, die den Objekttyp von Objekten in einem Amazon S3 S3-Bucket zurückgibt.
3. Konfigurieren Sie einen Lambda-Auslöser, der Ihre Funktion aufruft, wenn Objekte in Ihren Bucket hochgeladen werden.
4. Testen Sie Ihre Funktion, zuerst mit einem Dummy-Ereignis und dann mit dem Auslöser.

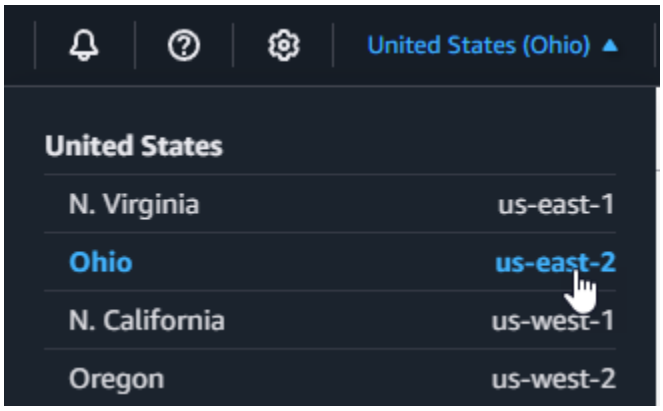
Wenn Sie diese Schritte ausführen, erfahren Sie, wie Sie eine Lambda-Ereignisfunktion so konfigurieren, dass sie immer dann ausgeführt wird, wenn Objekte zu einem Amazon S3-Bucket hinzugefügt oder aus einem Amazon S3 S3-Bucket gelöscht werden. Sie können dieses Tutorial abschließen, indem Sie nur die AWS-Managementkonsole verwenden.

Erstellen Sie einen Amazon S3 S3-Bucket



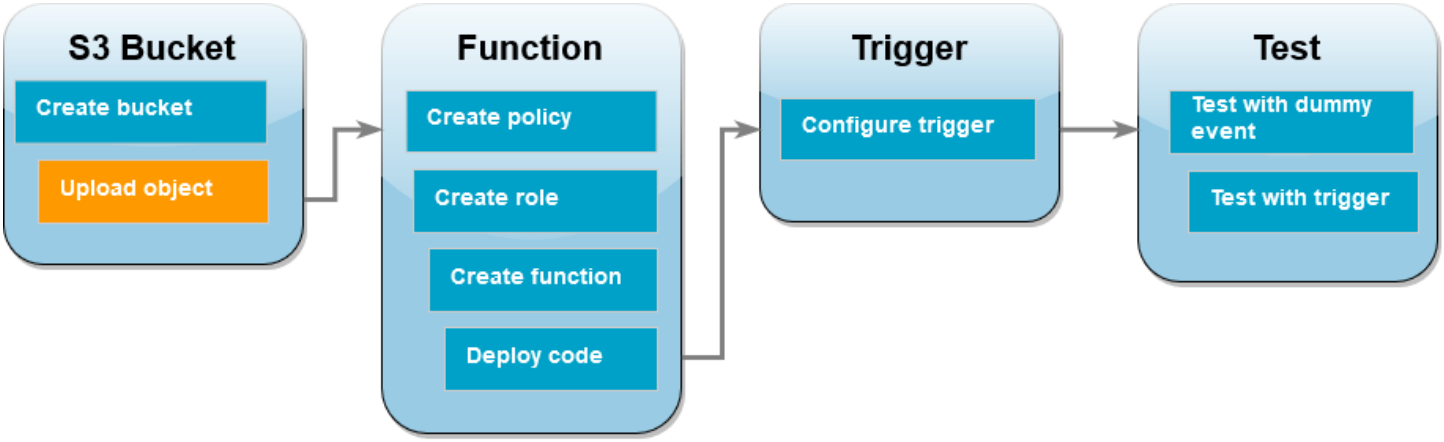
So erstellen Sie einen Amazon-S3-Bucket

1. Öffnen Sie die Amazon-S3-Konsole und wählen Sie die Seite Allzweck-Buckets aus.
2. Wählen Sie den, der Ihrem geografischen Standort AWS-Region am nächsten liegt. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern. Später im Tutorial müssen Sie Ihre Lambda-Ereignisfunktion in derselben Region erstellen.



3. Wählen Sie Create Bucket (Bucket erstellen) aus.
4. Führen Sie unter Allgemeine Konfiguration die folgenden Schritte aus:
   a. Stellen Sie sicher, dass für Bucket-Typ die Option Allzweck ausgewählt ist.
   b. Geben Sie für den Bucket-Namen einen global eindeutigen Namen ein, der den Regeln für die Bucket-Benennung von Amazon S3 entspricht. Bucket-Namen dürfen nur aus Kleinbuchstaben, Zahlen, Punkten (.) und Bindestrichen (-) bestehen.
5. Belassen Sie alle anderen Optionen auf ihren Standardwerten und wählen Sie Bucket erstellen aus.

Hochladen eines Testobjekts in Ihren Bucket

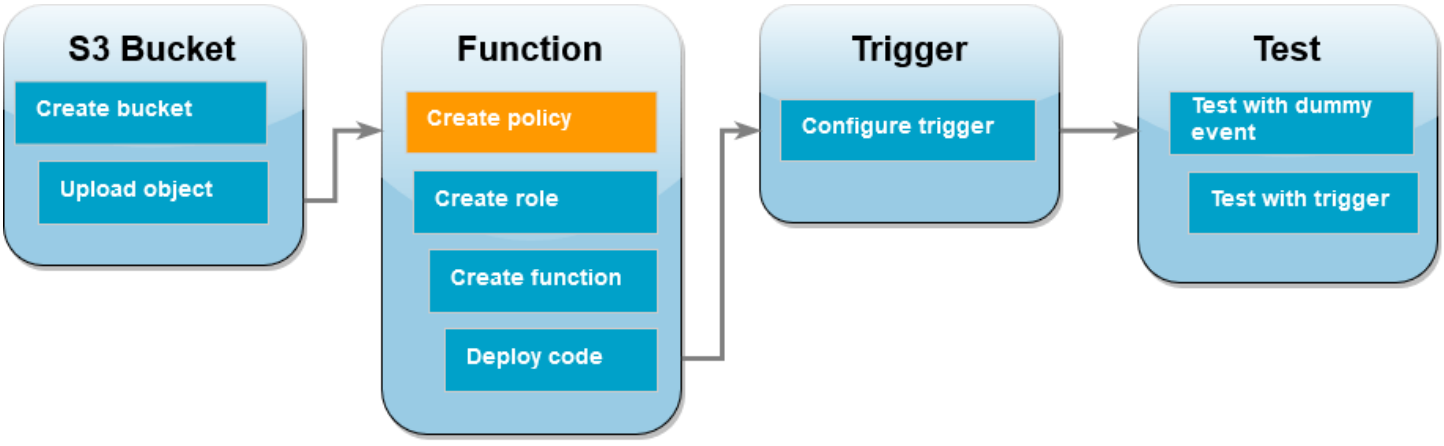


Hochladen eines Testobjekts

1. Öffnen Sie die Buckets-Seite der Amazon-S3-Konsole und wählen Sie den Bucket aus, den Sie im vorherigen Schritt erstellt haben.
2. Klicken Sie auf Hochladen.
3. Wählen Sie Dateien hinzufügen aus und wählen Sie das Objekt, das Sie hochladen möchten. Sie können eine beliebige Datei auswählen (z. B. `HappyFace.jpg`).
4. Wählen Sie Öffnen und anschließend Hochladen aus.

Später im Tutorial testen Sie Ihre Lambda-Ereignisfunktion mit diesem Objekt.

Erstellen einer Berechtigungsrichtlinie



Erstellen Sie eine Berechtigungsrichtlinie, die es Lambda ermöglicht, Objekte aus einem Amazon S3 S3-Bucket abzurufen und in Amazon CloudWatch Logs zu schreiben.

So erstellen Sie die Richtlinie

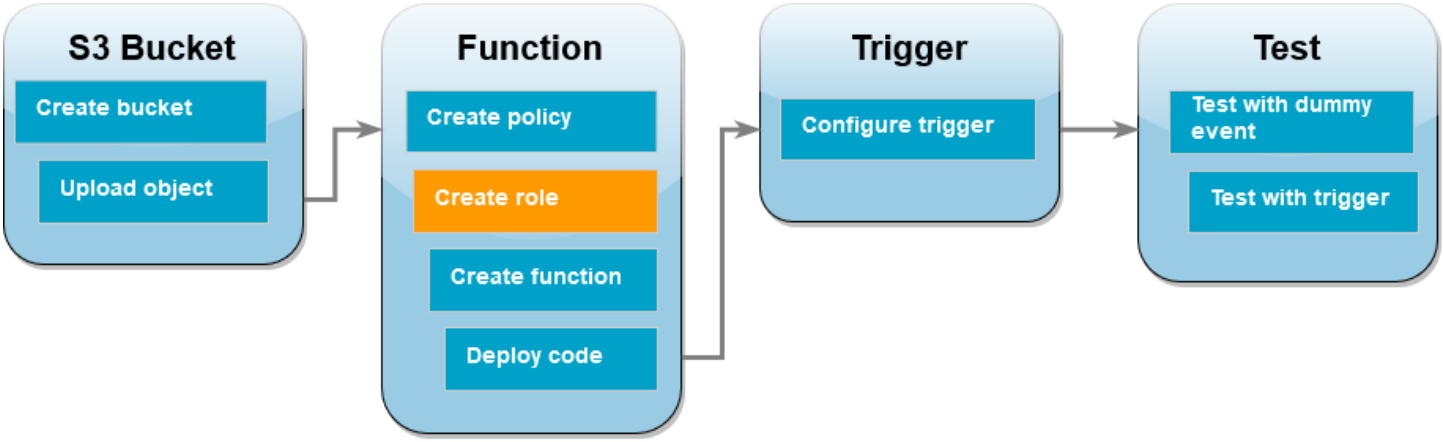
1. Öffnen Sie die Seite Richtlinien in der IAM-Konsole.
2. Wählen Sie Richtlinie erstellen aus.
3. Wählen Sie die Registerkarte JSON aus und kopieren Sie dann die folgende benutzerdefinierte JSON-Richtlinie in den JSON-Editor.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. Wählen Sie Next: Tags (Weiter: Tags) aus.
5. Klicken Sie auf Weiter: Prüfen.
6. Geben Sie unter Review policy (Richtlinie prüfen) für den Richtlinien-Namen **s3-trigger-tutorial** ein.
7. Wählen Sie Richtlinie erstellen aus.

Erstellen einer Ausführungsrolle

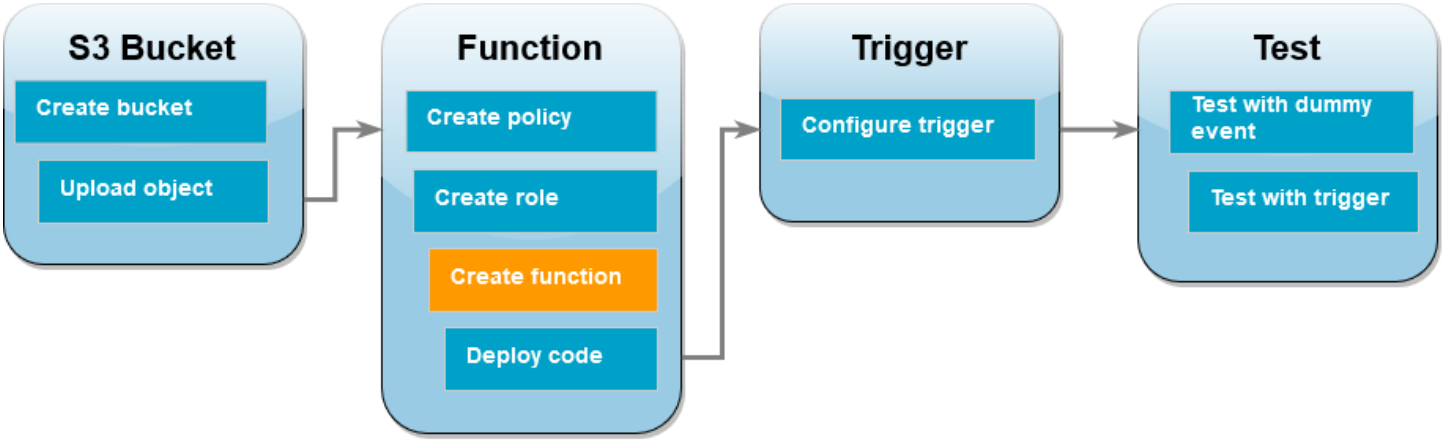


Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Ereignisfunktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. In diesem Schritt erstellen Sie eine Ausführungsrolle unter Verwendung der Berechtigungsrichtlinie, die Sie im vorherigen Schritt erstellt haben.

So erstellen Sie eine Ausführungsrolle und fügen Ihre benutzerdefinierte Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie als Typ der vertrauenswürdigen Entität AWS -Service und dann als Anwendungsfall Lambda aus.
4. Wählen Sie Weiter aus.
5. Geben Sie im Feld für die Richtliniensuche **`s3-trigger-tutorial`** ein.
6. Wählen Sie in den Suchergebnissen die von Ihnen erstellte Richtlinie (`s3-trigger-tutorial`) und dann die Option Next (Weiter) aus.
7. Geben Sie unter Role details (Rollendetails) für den Role name (Rollennamen) **`lambda-s3-trigger-role`** ein und wählen Sie dann Create role (Rolle erstellen) aus.

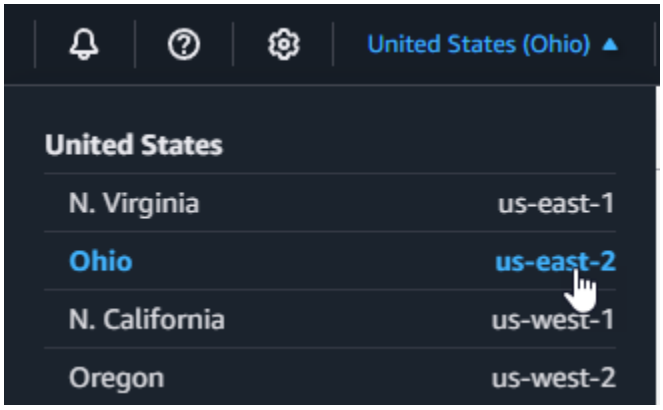
Erstellen Sie die Lambda-Ereignisfunktion



Erstellen Sie mit der Python 3.14-Laufzeit eine Lambda-Ereignisfunktion in der Konsole.

So erstellen Sie die Lambda-Ereignisfunktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Stellen Sie sicher, dass Sie in demselben Modus arbeiten, in dem AWS-Region Sie Ihren Amazon S3 S3-Bucket erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste oben auf dem Bildschirm ändern.



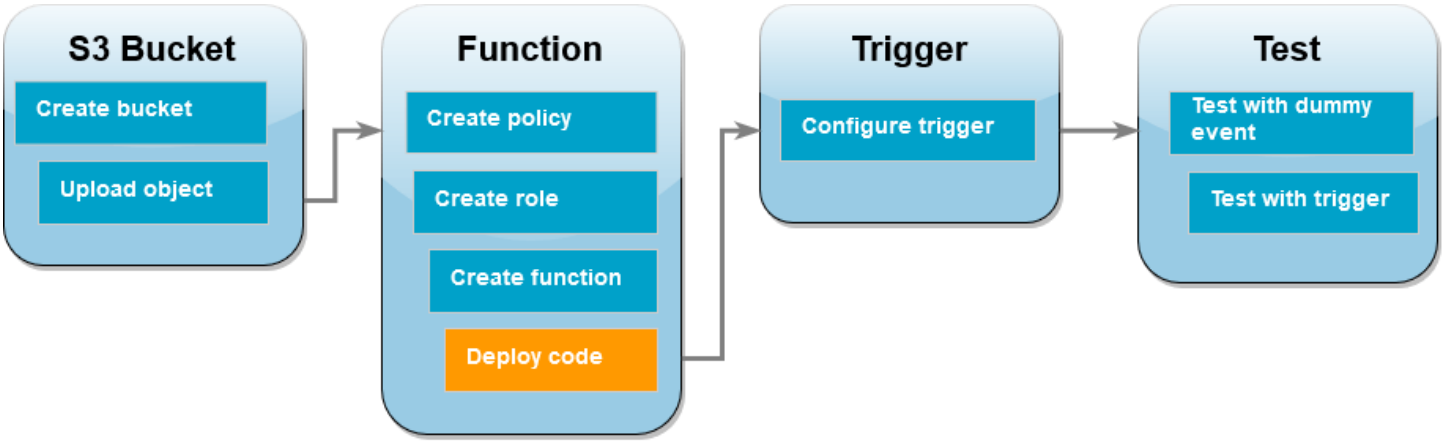
3. Wählen Sie „Event-Funktion erstellen“.
4. Wählen Sie Ohne Vorgabe erstellen aus.
5. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie unter Funktionsname `s3-trigger-tutorial` ein.
   b. Wählen Sie für Runtime Python 3.14.
   c. Wählen Sie für Architektur x86_64 aus.
6. Gehen Sie auf der Registerkarte Standard-Ausführungsrolle ändern wie folgt vor:

a. Erweitern Sie die Registerkarte und wählen Sie dann Verwenden einer vorhandenen Rolle aus.

b. Wählen Sie die zuvor erstellte `lambda-s3-trigger-role` aus.

7. Wählen Sie „Ereignisfunktion erstellen“.

Stellen Sie den Code für die Ereignisfunktion bereit



Dieses Tutorial verwendet die Python 3.14-Laufzeit, aber wir haben auch Beispielcodedateien für andere Laufzeiten bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um den Code für die Laufzeit zu sehen, an der Sie interessiert sind.

Die Lambda-Ereignisfunktion ruft den Schlüsselnamen des hochgeladenen Objekts und den Namen des Buckets aus dem `event` Parameter ab, den sie von Amazon S3 erhält. Die Funktion verwendet dann die Methode get_object von, AWS SDK für Python (Boto3) um die Metadaten des Objekts abzurufen, einschließlich des Inhaltstyps (MIME-Typ) des hochgeladenen Objekts.

Um den Code für die Ereignisfunktion bereitzustellen

1. Wählen Sie im folgenden Feld die Registerkarte Python und kopieren Sie den Code.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von .NET

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }
```

```
        public async Task<string> Handler(S3Event evt, ILambdaContext
 context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key =
 HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
 key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Go

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
```

```
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Java

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines S3-Ereignisses mit Lambda unter Verwendung JavaScript.

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

Konsumieren eines S3-Ereignisses mit Lambda unter Verwendung TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());
```

```
            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Python

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')
```

```
def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Ruby.

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
```

```
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

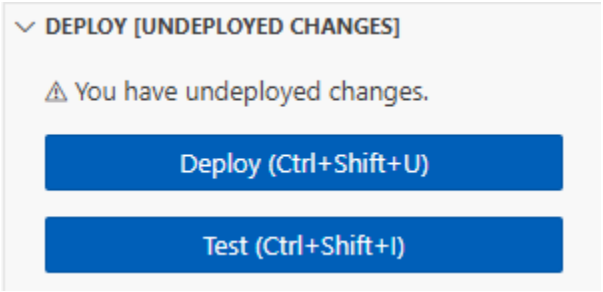
Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Rust

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
```

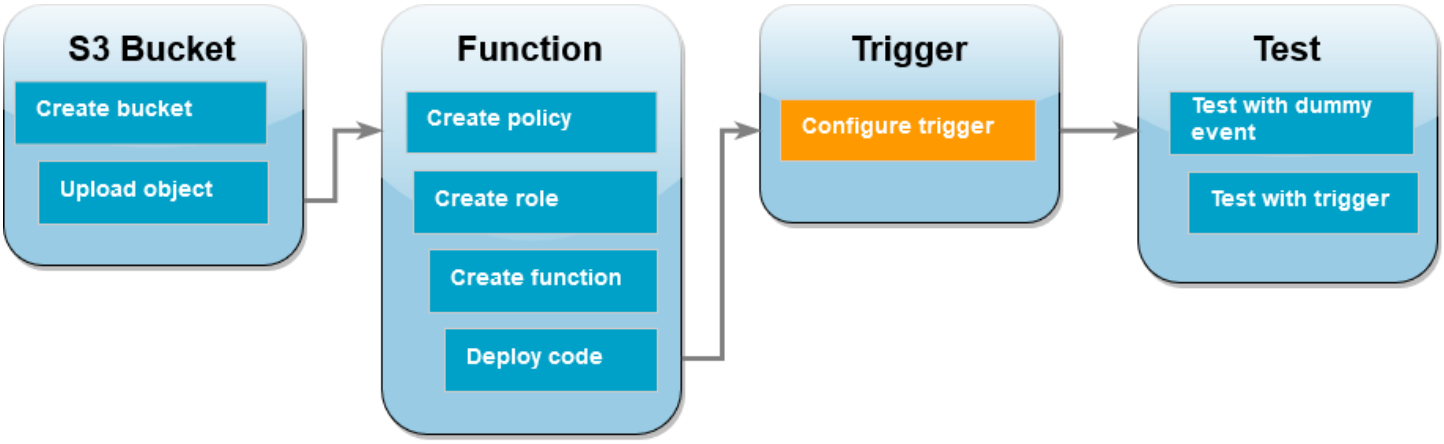
```
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
```

```
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

2. Fügen Sie den Code im Code-Quellbereich der Lambda-Konsole in den Code-Editor ein und ersetzen Sie dabei den von Lambda erstellten Code.
3. Wählen Sie im Abschnitt DEPLOY die Option Deploy aus, um den Code Ihrer Funktion zu aktualisieren:

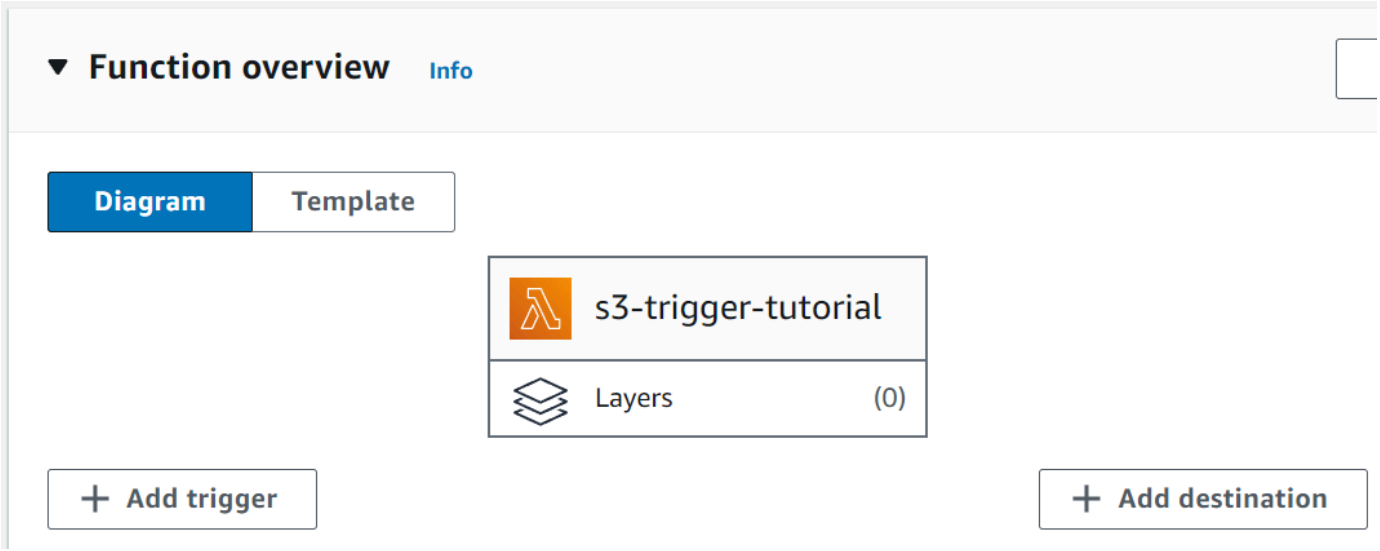


Erstellen des Amazon-S3-Auslösers



Erstellen des Amazon-S3-Auslösers

1. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).



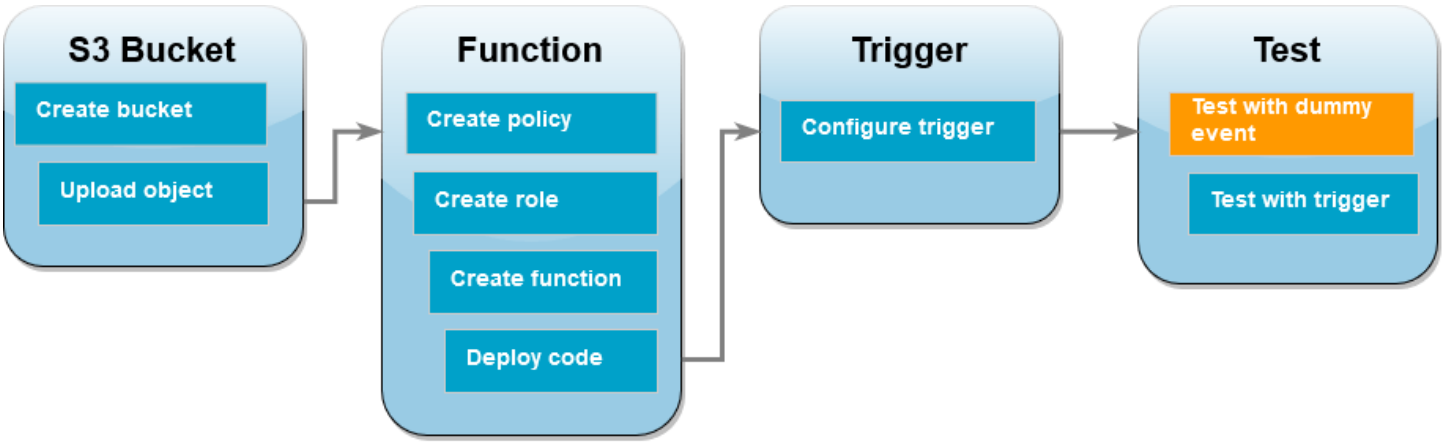
2. Wählen Sie S3 aus.
3. Wählen Sie unter Bucket den Bucket aus, den Sie zuvor im Tutorial erstellt haben.
4. Vergewissern Sie sich, dass unter Ereignistypen die Option Alle Ereignisse zur Objekterstellung ausgewählt ist.
5. Aktivieren Sie unter Rekursiver Aufruf das Kontrollkästchen, um zu bestätigen, dass die Verwendung desselben Amazon-S3-Buckets für die Ein- und Ausgabe nicht empfohlen wird.
6. Wählen Sie Hinzufügen aus.

> **Note**
>
> Wenn Sie mithilfe der Lambda-Konsole einen Amazon S3-Trigger für eine Lambda-Ereignisfunktion erstellen, konfiguriert Amazon S3 eine Ereignisbenachrichtigung für den von Ihnen angegebenen Bucket. Bevor diese Ereignisbenachrichtigung konfiguriert wird, führt Amazon S3 eine Reihe von Prüfungen durch, um zu bestätigen, dass das Ereignisziel existiert und die erforderlichen IAM-Richtlinien hat. Amazon S3 führt diese Tests auch bei allen anderen für diesen Bucket konfigurierten Ereignisbenachrichtigungen durch.
> Wenn der Bucket zuvor Ereignisziele für Ressourcen konfiguriert hat, die nicht mehr existieren oder für Ressourcen, die nicht über die erforderlichen Berechtigungsrichtlinien verfügen, kann Amazon S3 aufgrund dieser Prüfung keine neue Ereignisbenachrichtigung erstellen. Es wird die folgende Fehlermeldung angezeigt, die besagt, dass Ihr Auslöser nicht erstellt werden konnte:

```
An error occurred when creating the trigger: Unable to validate the following
 destination configurations.
```

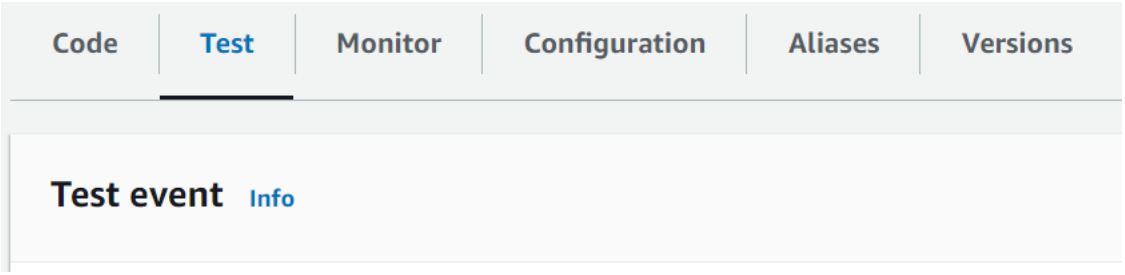
Sie können diesen Fehler sehen, wenn Sie zuvor einen Trigger für eine andere Lambda-Ereignisfunktion konfiguriert haben, die denselben Bucket verwendet, und Sie die Funktion inzwischen gelöscht oder ihre Berechtigungsrichtlinien geändert haben.

Testen Sie Ihre Lambda-Event-Funktion mit einem Dummy-Event



Um die Lambda-Ereignisfunktion mit einem Dummy-Ereignis zu testen

1. Wählen Sie auf der Konsolenseite für Ihre Funktion die Registerkarte Test aus.



2. Geben Sie für Event name (Ereignisname) `MyTestEvent` ein.
3. Fügen Sie im Event-JSON das folgende Testereignis ein. Achten Sie darauf, diese Werte zu ersetzen:

   - Ersetzen Sie `us-east-1` durch die Region, in der Sie den Amazon-S3-Bucket erstellt haben.
   - Ersetzen Sie beide Vorkommen von `amzn-s3-demo-bucket` durch den Namen Ihres eigenen Amazon-S3-Buckets.

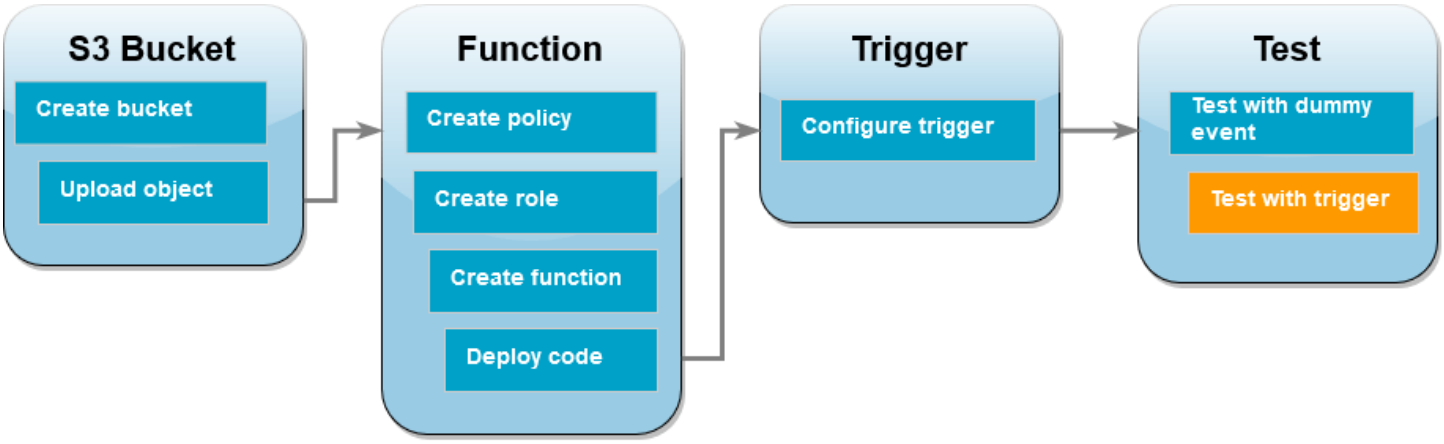
- Ersetzen Sie `test%2FKey` durch den Namen des Testobjekts, das Sie zuvor in Ihren Bucket hochgeladen haben (z. B.`HappyFace.jpg`).

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

4. Wählen Sie Speichern.
5. Wählen Sie Test aus.
6. Wenn Ihre Funktion erfolgreich ausgeführt wird, sehen Sie auf der Registerkarte Ausführungsergebnisse eine Ausgabe, die der folgenden ähnelt.

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'amzn-s3-demo-bucket', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

Testen Sie die Lambda-Ereignisfunktion mit dem Amazon S3 S3-Trigger



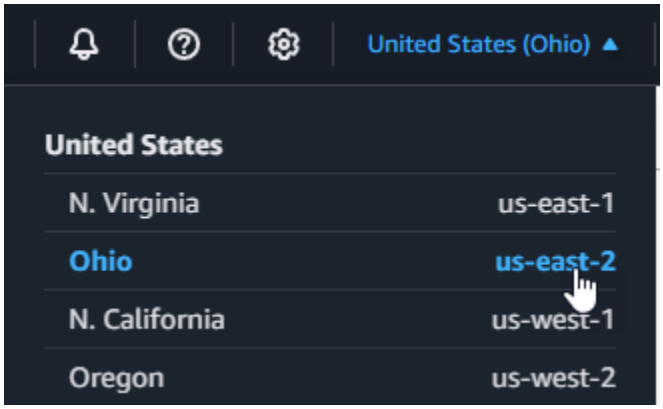
Um Ihre Funktion mit dem konfigurierten Auslöser zu testen, können Sie unter Verwendung der Konsole ein Objekt in Ihren Amazon-S3-Bucket hochladen. Um zu überprüfen, ob Ihre Lambda-Ereignisfunktion wie erwartet ausgeführt wurde, verwenden Sie CloudWatch Logs, um die Ausgabe Ihrer Funktion einzusehen.

So laden Sie Objekte in Ihren Amazon-S3-Bucket hoch

1. Öffnen Sie die Buckets-Seite der Amazon-S3-Konsole und wählen Sie den Bucket aus, den Sie zuvor erstellt haben.
2. Klicken Sie auf Upload.
3. Wählen Sie Dateien hinzufügen aus und wählen Sie mit der Dateiauswahl ein Objekt aus, das Sie hochladen möchten. Bei diesem Objekt kann es sich um eine beliebige von Ihnen ausgewählte Datei handeln.
4. Wählen Sie Öffnen und anschließend Hochladen aus.

Um den Aufruf der Event-Funktion mithilfe von Logs zu überprüfen CloudWatch

1. Öffnen Sie die CloudWatch-Konsole.
2. Stellen Sie sicher, dass Sie in derselben Umgebung arbeiten, in der AWS-Region Sie Ihre Lambda-Ereignisfunktion erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste oben auf dem Bildschirm ändern.



3. Wählen Sie Protokolle und anschließend Protokollgruppen aus.
4. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/s3-trigger-tutorial`) aus.
5. Wählen Sie im Bereich Protokollstreams den neuesten Protokollstream aus.
6. Wenn Ihre Funktion als Reaktion auf Ihren Amazon S3 S3-Trigger korrekt aufgerufen wurde, erhalten Sie eine Ausgabe, die der folgenden ähnelt. Welchen `CONTENT TYPE` Sie sehen, hängt von der Art der Datei ab, die Sie in Ihren Bucket hochgeladen haben.

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, verhindern Sie unnötige Gebühren für Ihre AWS-Konto.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie den S3-Bucket:

1. Öffnen Sie die Amazon S3-Konsole.
2. Wählen Sie den Bucket aus, den Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen des Buckets in das Texteingabefeld ein.
5. Wählen Sie Bucket löschen aus.

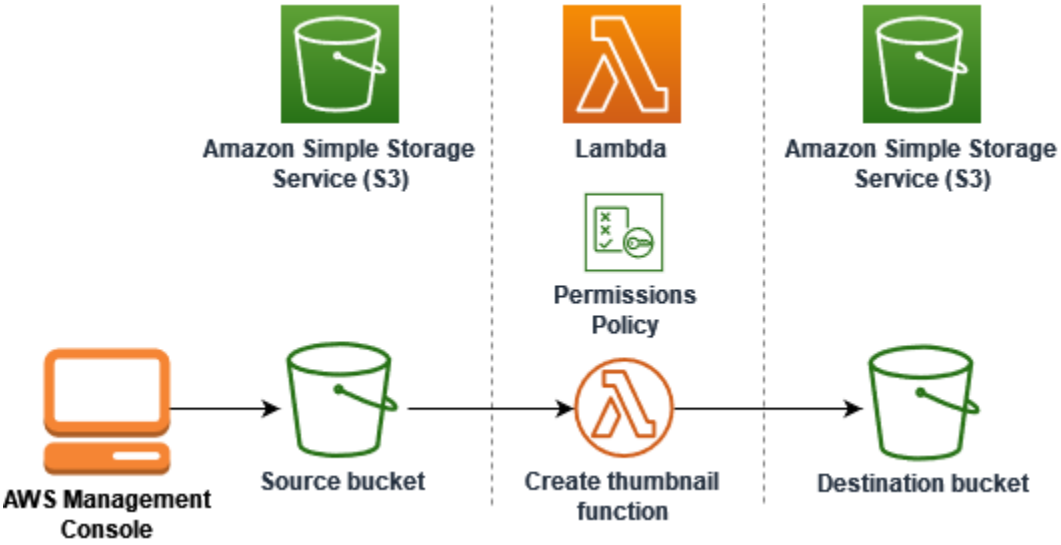
Nächste Schritte

In Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern ruft der Amazon-S3-Auslöser eine Funktion auf, die für jede Bilddatei, die in einen Bucket hochgeladen wird, ein Miniaturbild erstellt. Dieses Tutorial erfordert ein gewisses Maß AWS an Wissen über Lambda-Domänen. Es zeigt, wie Ressourcen mithilfe von AWS Command Line Interface (AWS CLI)

erstellt werden und wie ein Bereitstellungspaket für das ZIP-Dateiarchiv für die Funktion und ihre Abhängigkeiten erstellt wird.

Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern

In diesem Tutorial erstellen und konfigurieren Sie eine Lambda-Ereignisfunktion, die die Größe von Bildern ändert, die einem Amazon Simple Storage Service (Amazon S3) -Bucket hinzugefügt wurden. Wenn Sie Ihrem Bucket eine Image-Datei hinzufügen, ruft Amazon S3 Ihre Lambda-Ereignisfunktion auf. Die Funktion erstellt daraufhin eine Miniaturversion des Bilds und gibt sie an einen anderen Amazon-S3-Bucket aus.



Führen Sie für dieses Tutorial die folgenden Schritte aus:

1. Erstellen Sie Quell- und Ziel-Buckets für Amazon S3 und laden Sie ein Beispielbild hoch.
2. Erstellen Sie eine Lambda-Ereignisfunktion, die die Größe eines Bilds ändert und eine Miniaturansicht an einen Amazon S3 S3-Bucket ausgibt.
3. Konfigurieren Sie einen Lambda-Auslöser, der Ihre Funktion aufruft, wenn Objekte in Ihren Quell-Bucket hochgeladen werden.
4. Testen Sie Ihre Funktion zunächst mit einem Dummy-Ereignis und anschließend durch Hochladen eines Bilds in Ihren Quell-Bucket.

Wenn Sie diese Schritte ausführen, lernen Sie, wie Sie Lambda verwenden, um eine Dateiverarbeitungsaufgabe für Objekte auszuführen, die zu einem Amazon S3 S3-Bucket hinzugefügt wurden. Sie können dieses Tutorial mit dem AWS Command Line Interface (AWS CLI) oder dem AWS-Managementkonsole abschließen.

Ein einfacheres Beispiel, das zeigt, wie Sie einen Amazon-S3-Auslöser für Lambda konfigurieren, finden Sie unter Tutorial: Verwenden eines Amazon-S3-Auslösers zum Aufrufen einer Lambda-Funktion.

Themen

- Voraussetzungen
- Erstellen von zwei Amazon-S3-Buckets
- Hochladen eines Testbilds in Ihren Quell-Bucket
- Erstellen einer Berechtigungsrichtlinie
- Erstellen einer Ausführungsrolle
- Erstellen Sie das Event Function-Bereitstellungspaket
- Erstellen Sie die Lambda-Ereignisfunktion
- Amazon S3 so konfigurieren, dass die Event-Funktion aufgerufen wird
- Testen Sie Ihre Lambda-Event-Funktion mit einem Dummy-Event
- Testen Sie Ihre Event-Funktion mit dem Amazon S3 S3-Trigger
- Bereinigen Ihrer Ressourcen

Voraussetzungen

Wenn Sie den verwenden möchten, AWS CLI um das Tutorial abzuschließen, installieren Sie die neueste Version von AWS Command Line Interface.

Für Ihren Lambda-Ereignisfunktionscode können Sie Python oder Node.js verwenden. Installieren Sie die Tools zur Sprachunterstützung und einen Paketmanager für die gewünschte Sprache.

Installieren Sie das AWS Command Line Interface

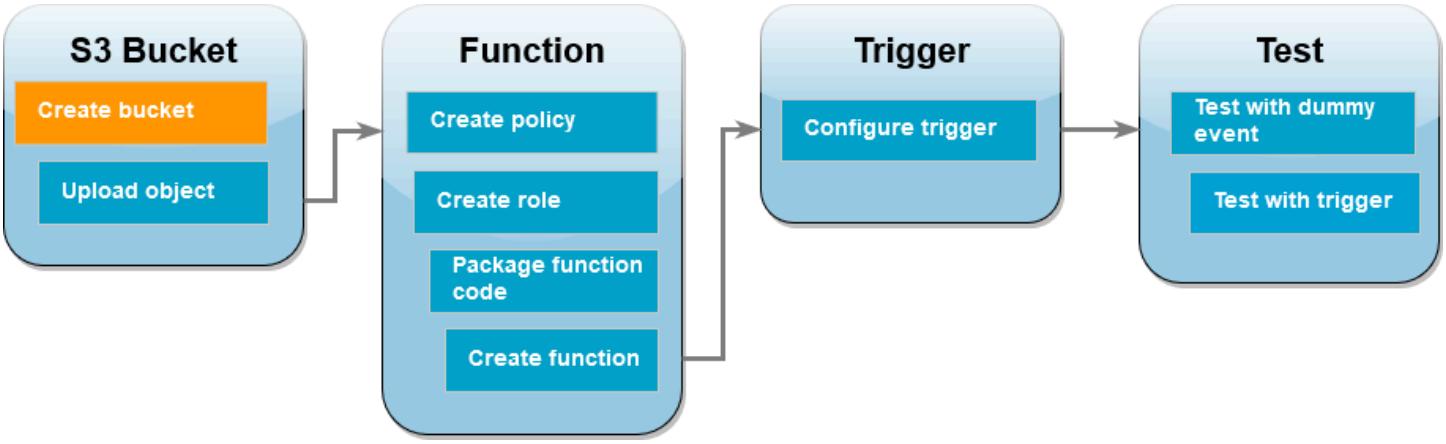
Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine

Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen von zwei Amazon-S3-Buckets

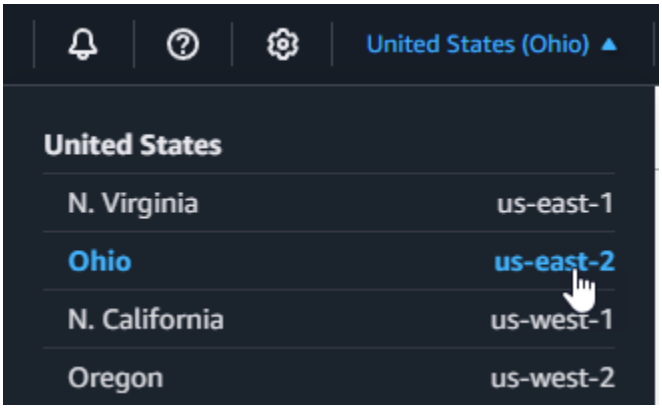


Erstellen Sie zunächst zwei Amazon-S3-Buckets. Der erste Bucket ist der Quell-Bucket, in den Sie Ihre Bilder hochladen. Der zweite Bucket wird von Lambda verwendet, um das verkleinerte Miniaturbild zu speichern, wenn Sie Ihre Funktion aufrufen.

AWS-Managementkonsole

So erstellen Sie die Amazon-S3-Buckets (Konsole)

1. Öffnen Sie die Amazon-S3-Konsole und wählen Sie die Seite Allzweck-Buckets aus.
2. Wählen Sie den, der Ihrem geografischen Standort AWS-Region am nächsten liegt. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern. Später im Tutorial müssen Sie Ihre Lambda-Ereignisfunktion in derselben Region erstellen.



3. Wählen Sie Create Bucket (Bucket erstellen) aus.

4. Führen Sie unter Allgemeine Konfiguration die folgenden Schritte aus:

   a. Stellen Sie sicher, dass für Bucket-Typ die Option Allzweck ausgewählt ist.

   b. Geben Sie für den Bucket-Namen einen global eindeutigen Namen ein, der den Regeln für die Bucket-Benennung von Amazon S3 entspricht. Bucket-Namen dürfen nur aus Kleinbuchstaben, Zahlen, Punkten (.) und Bindestrichen (-) bestehen.

5. Belassen Sie alle anderen Optionen auf ihren Standardwerten und wählen Sie Bucket erstellen aus.

6. Wiederholen Sie die Schritte 1 bis 5, um Ihren Ziel-Bucket zu erstellen. Geben Sie unter Bucket-Name den Namen **`amzn-s3-demo-source-bucket-resized`** ein. **`amzn-s3-demo-source-bucket`** ist hierbei der Name des Quell-Buckets, den Sie soeben erstellt haben.

AWS CLI

Um die Amazon S3 S3-Buckets zu erstellen (AWS CLI)

1. Führen Sie den folgenden CLI-Befehl aus, um Ihren Quell-Bucket zu erstellen. Der gewählte Name für Ihren Bucket muss global eindeutig sein und den Regeln für die Benennung von Buckets für Amazon S3 entsprechen. Namen dürfen nur Kleinbuchstaben, Zahlen, Punkte (.) und Bindestriche (-) enthalten. Wählen Sie für `region` und `LocationConstraint` die AWS-Region aus, die Ihrem geografischen Standort am nächsten ist.

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-source-bucket --region us-east-1 \
   --create-bucket-configuration LocationConstraint=us-east-1
   ```

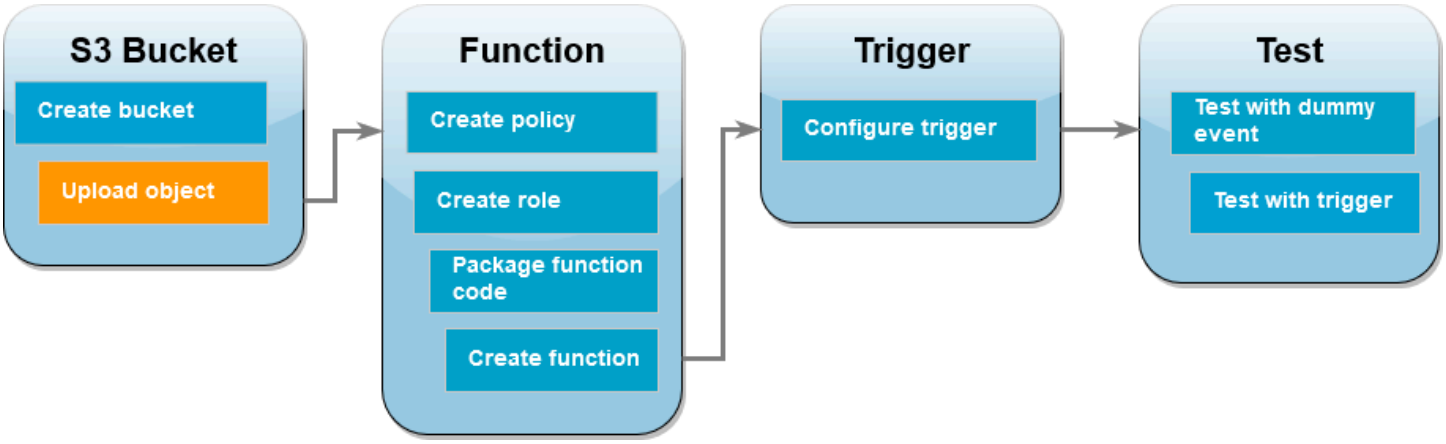
   Später im Tutorial müssen Sie Ihre Lambda-Ereignisfunktion im selben Verzeichnis AWS-Region wie in Ihrem Quell-Bucket erstellen. Notieren Sie sich also die Region, die Sie ausgewählt haben.

2. Führen Sie den folgenden Befehl aus, um Ihren Ziel-Bucket zu erstellen. Verwenden Sie für den Bucket-Namen **`amzn-s3-demo-source-bucket-resized`**. **`amzn-s3-demo-source-bucket`** ist hierbei der Name des Quell-Buckets, den Sie in Schritt 1 erstellt haben. Wählen Sie für `region` und dasselbe aus`LocationConstraint`, das AWS-Region Sie zur Erstellung Ihres Quell-Buckets verwendet haben.

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-source-bucket-resized --region us-
   east-1 \
   ```

```
--create-bucket-configuration LocationConstraint=us-east-1
```

Hochladen eines Testbilds in Ihren Quell-Bucket



Später im Tutorial testen Sie Ihre Lambda-Ereignisfunktion, indem Sie sie mit der AWS CLI oder der Lambda-Konsole aufrufen. Damit Sie sich vergewissern können, dass Ihre Funktion ordnungsgemäß funktioniert, muss Ihr Quell-Bucket ein Testbild enthalten. Bei diesem Bild kann es sich um eine beliebige JPG- oder PNG-Datei handeln.

AWS-Managementkonsole

So laden Sie ein Testbild in Ihren Quell-Bucket hoch (Konsole)

1. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole.
2. Wählen Sie den Quell-Bucket aus, den Sie im vorherigen Schritt erstellt haben.
3. Klicken Sie auf Hochladen.
4. Wählen Sie Dateien hinzufügen aus und wählen Sie mit der Dateiauswahl das Objekt aus, das Sie hochladen möchten.
5. Wählen Sie Öffnen und anschließend Hochladen aus.

AWS CLI

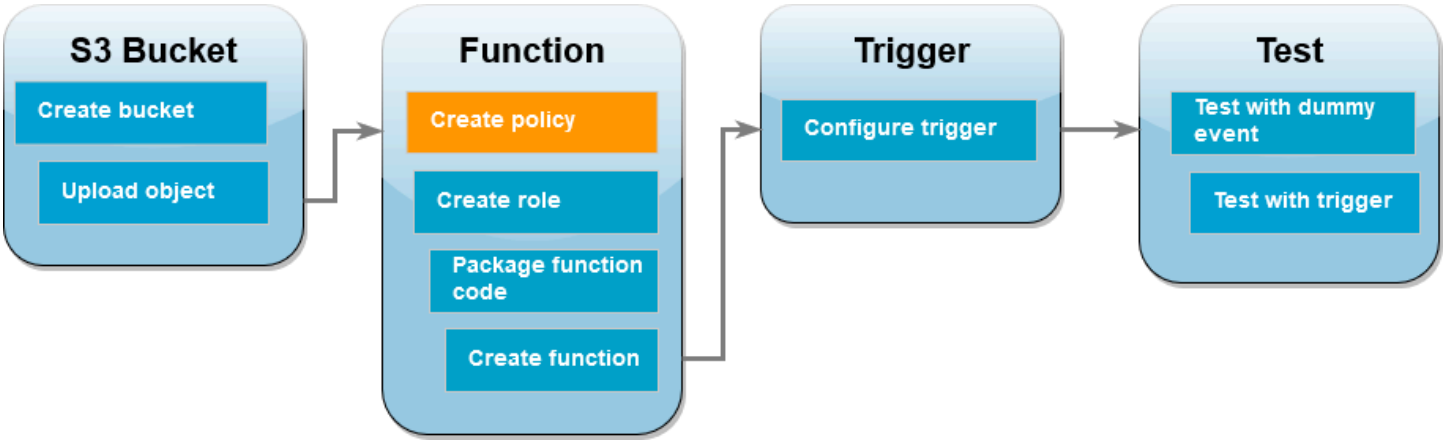
Um ein Testbild in Ihren Quell-Bucket hochzuladen (AWS CLI)

- Führen Sie in dem Verzeichnis, das das hochzuladende Bild enthält, den folgenden CLI-Befehl aus. Ersetzen Sie dabei den Parameter `--bucket` durch den Namen Ihres Quell-

Buckets. Verwenden Sie für die Parameter `--key` und `--body` den Dateinamen Ihres Testbilds.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key HappyFace.jpg --
body ./HappyFace.jpg
```

Erstellen einer Berechtigungsrichtlinie



Der erste Schritt bei der Erstellung Ihrer Lambda-Ereignisfunktion besteht darin, eine Berechtigungsrichtlinie zu erstellen. Diese Richtlinie gibt Ihrer Funktion die Berechtigungen, die sie für den Zugriff auf andere AWS Ressourcen benötigt. In diesem Tutorial gewährt die Richtlinie Lambda Lese- und Schreibberechtigungen für Amazon S3 S3-Buckets und ermöglicht es ihm, in Amazon CloudWatch Logs zu schreiben.

AWS-Managementkonsole

So erstellen Sie die Richtlinie (Konsole)

1. Öffnen Sie die Seite Richtlinien der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie Richtlinie erstellen aus.
3. Wählen Sie die Registerkarte JSON aus und kopieren Sie dann die folgende benutzerdefinierte JSON-Richtlinie in den JSON-Editor.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. Wählen Sie Weiter aus.
5. Geben Sie unter Richtliniendetails für den Richtliniennamen **`LambdaS3Policy`** ein.
6. Wählen Sie Richtlinie erstellen aus.

AWS CLI

Um die Richtlinie zu erstellen (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `policy.json`:

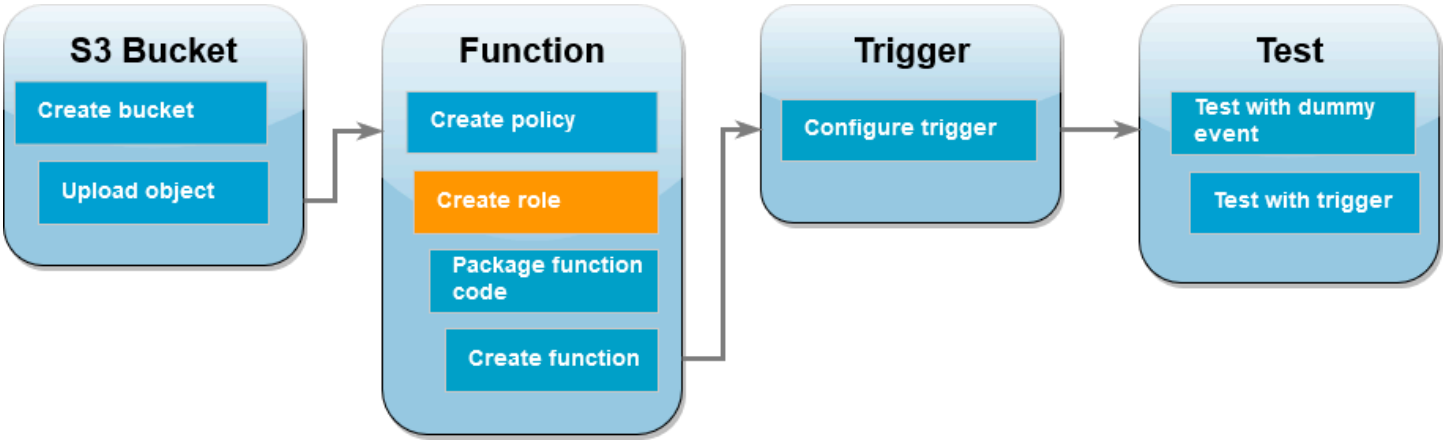
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
```

```
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. Führen Sie in dem Verzeichnis, in dem Sie das JSON-Richtliniendokument gespeichert haben, den folgenden CLI-Befehl aus:

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

Erstellen einer Ausführungsrolle



Eine Ausführungsrolle ist eine IAM-Rolle, die einer Lambda-Ereignisfunktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. Um Ihrer Funktion Lese- und Schreibzugriff auf einen Amazon-S3-Bucket zu erteilen, fügen Sie die Berechtigungsrichtlinie an, die Sie im vorherigen Schritt erstellt haben.

AWS-Managementkonsole

So erstellen Sie eine Ausführungsrolle und fügen Ihre Berechtigungsrichtlinie hinzu (Konsole)

1. Öffnen Sie die Seite Rollen der (IAM-)Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS-Service und unter Anwendungsfall die Option Lambda aus.
4. Wählen Sie Weiter aus.
5. Fügen Sie die Berechtigungsrichtlinie hinzu, die Sie im vorherigen Schritt erstellt haben:
   a. Geben Sie im Feld für die Richtliniensuche **`LambdaS3Policy`** ein.
   b. Aktivieren Sie in den Suchergebnissen das Kontrollkästchen für `LambdaS3Policy`.
   c. Wählen Sie Weiter aus.
6. Geben Sie unter Rollendetails im Feld Rollenname den Namen **`LambdaS3Role`** ein.
7. Wählen Sie Rolle erstellen aus.

AWS CLI

Um eine Ausführungsrolle zu erstellen und Ihre Berechtigungsrichtlinie anzuhängen (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `trust-policy.json`: Diese Vertrauensrichtlinie ermöglicht es Lambda, die Berechtigungen der Rolle zu verwenden, indem dem Dienstprinzipal die `lambda.amazonaws.com` Erlaubnis erteilt wird, die `AssumeRole` Aktion AWS -Security-Token-Service (AWS STS) aufzurufen.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
```

```
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

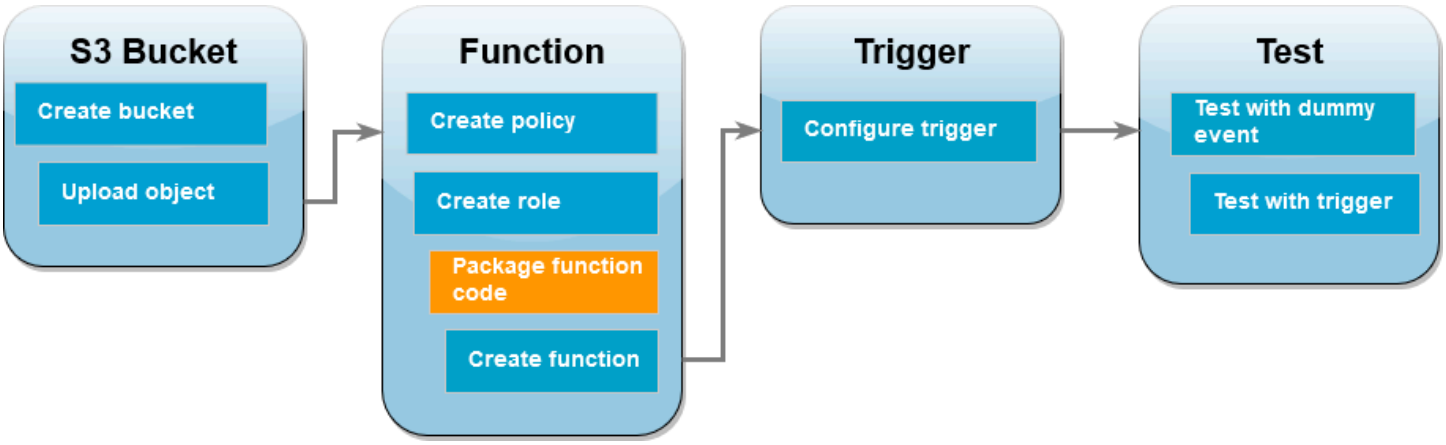
2. Führen Sie in dem Verzeichnis, in dem Sie das JSON-Vertrauensrichtliniendokument gespeichert haben, den folgenden CLI-Befehl aus, um die Ausführungsrolle zu erstellen:

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. Führen Sie den folgenden CLI-Befehl aus, um die im vorherigen Schritt erstellte Berechtigungsrolle anzufügen. Ersetzen Sie die AWS-Konto Nummer im ARN der Police durch Ihre eigene Kontonummer.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

Erstellen Sie das Event Function-Bereitstellungspaket



Um Ihre Funktion zu erstellen, erstellen Sie ein Bereitstellungspaket, das Ihren Funktionscode und die zugehörigen Abhängigkeiten enthält. Bei der hier verwendeten Funktion `CreateThumbnail` verwendet Ihr Funktionscode eine separate Bibliothek für die Anpassung der Bildgröße. Folgen Sie den Anweisungen für die von Ihnen gewählte Sprache, um ein Bereitstellungspaket zu erstellen, das die erforderliche Bibliothek enthält.

Node.js

Um das Bereitstellungspaket zu erstellen (Node.js)

1. Erstellen Sie ein Verzeichnis namens `lambda-s3` für Ihren Funktionscode und die Abhängigkeiten und navigieren Sie dorthin.

```
mkdir lambda-s3
cd lambda-s3
```

2. Erstellen Sie ein neues Node.js Projekt mit`npm`. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

```
npm init
```

3. Speichern Sie den folgenden Funktionscode in einer lokalen Datei namens `index.mjs`: Stellen Sie sicher, dass Sie `us-east-1` durch das AWS-Region ersetzen, in dem Sie Ihre eigenen Quell- und Ziel-Buckets erstellt haben.

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';


// create S3 client
const s3 = new S3Client({region: 'us-east-1'});

// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
```

```
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
    throw new Error('Unknown object stream type');
  }


} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;
```

```
// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
    Key: dstKey,
    Body: output_buffer,
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

4. Installieren Sie die Sharp-Bibliothek mithilfe von npm im Verzeichnis `lambda-s3`. Beachten Sie, dass die neueste Version von Sharp (0.33) nicht mit Lambda kompatibel ist. Installieren Sie für dieses Tutorial daher Version 0.32.6.

```
npm install sharp@0.32.6
```

Der npm-Befehl `install` erstellt ein Verzeichnis namens `node_modules` für Ihre Module. Nach diesem Schritt sollte Ihre Verzeichnisstruktur wie folgt aussehen:

```
lambda-s3
|- index.mjs
|- node_modules
```

```
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

5. Erstellen Sie ein ZIP-Bereitstellungspaket, das Ihren Funktionscode und die zugehörigen Abhängigkeiten enthält. Führen Sie unter macOS oder Linux den folgenden Befehl aus:

```
zip -r function.zip .
```

Verwenden Sie unter Windows Ihr bevorzugtes ZIP-Programm, um eine ZIP-Datei zu erstellen. Achten Sie darauf, dass sich die Dateien `index.mjs`, `package.json` und `package-lock.json` und das Verzeichnis `node_modules` im Stammverzeichnis der ZIP-Datei befinden.

Python

So erstellen Sie das Bereitstellungspaket (Python)

1. Speichern Sie den Beispielcode als Datei mit dem Namen `lambda_function.py`.

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
```

```
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. Erstellen Sie in dem Verzeichnis, in dem Sie auch Ihre `lambda_function.py`-Datei erstellt haben, ein neues Verzeichnis mit dem Namen `package` und installieren Sie die Bibliothek Pillow (PIL) sowie das AWS SDK für Python (Boto3). Die Lambda-Python-Laufzeit enthält zwar eine Version des Boto3 SDK, es empfiehlt sich jedoch, dem Bereitstellungspaket alle Abhängigkeiten Ihrer Funktion hinzuzufügen, auch wenn sie in der Laufzeit enthalten sind. Weitere Informationen finden Sie in der Laufzeitabhängigkeiten in Python.

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.12 \
--only-binary=:all: --upgrade \
pillow boto3
```

   Die Pillow-Bibliothek enthält C/C ++-Code. Mithilfe der `--only-binary=:all:` Optionen `--platform manylinux_2014_x86_64` und lädt Pip eine Version von Pillow herunter und installiert sie, die vorkompilierte Binärdateien enthält, die mit dem Amazon Linux 2-Betriebssystem kompatibel sind. Dadurch wird sichergestellt, dass Ihr Bereitstellungspaket in der Lambda-Ausführungsumgebung funktioniert, unabhängig vom Betriebssystem und der Architektur Ihrer lokalen Build-Maschine.

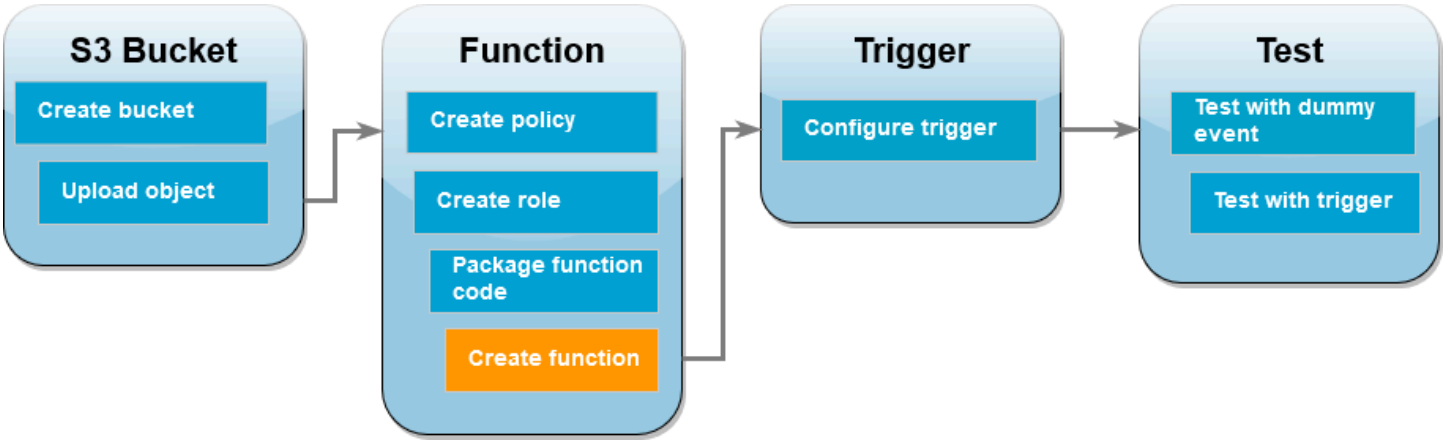
3. Erstellen Sie eine ZIP-Datei, die Ihren Anwendungscode sowie die Pillow- und die Boto3-Bibliothek enthält. Führen Sie unter Linux oder MacOS die folgenden Befehle über Ihre Befehlszeilenschnittstelle aus:

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

Verwenden Sie unter Windows Ihr bevorzugtes ZIP-Tool, um die `lambda_function.zip`-Datei zu erstellen. Achten Sie darauf, dass sich die Datei `lambda_function.py` und die Ordner, die Ihre Abhängigkeiten enthalten, im Stammverzeichnis der ZIP-Datei befinden.

Sie können Ihr Bereitstellungspaket auch in einer virtuellen Python-Umgebung erstellen. Siehe Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions

Erstellen Sie die Lambda-Ereignisfunktion



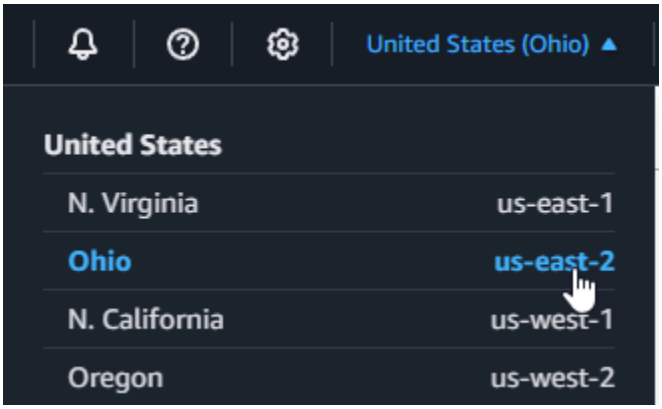
Sie können Ihre Lambda-Ereignisfunktion entweder mit der AWS CLI oder der Lambda-Konsole erstellen. Folgen Sie den Anweisungen für die von Ihnen gewählte Sprache, um die Funktion zu erstellen.

AWS-Managementkonsole

So erstellen Sie die Event-Funktion (Konsole)

Um Ihre Lambda-Event-Funktion mithilfe der Konsole zu erstellen, erstellen Sie zunächst eine Basisfunktion, die etwas „Hallo Welt“ -Code enthält. Anschließend ersetzen Sie diesen Code durch Ihren eigenen Funktionscode, indem Sie die im vorherigen Schritt erstellte ZIP- oder JAR-Datei hochladen.

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Stellen Sie sicher, dass Sie in demselben Modus arbeiten, in dem AWS-Region Sie Ihren Amazon S3 S3-Bucket erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern.



3. Wählen Sie „Event-Funktion erstellen“.
4. Wählen Sie Author from scratch aus.
5. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Function name (Funktionsname) **`CreateThumbnail`** ein.
   b. Wählen Sie für Runtime entweder Node.js 22.x oder Python 3.12, je nachdem, welche Sprache Sie für Ihre Funktion ausgewählt haben.
   c. Wählen Sie für Architektur x86_64 aus.
6. Gehen Sie auf der Registerkarte Standard-Ausführungsrolle ändern wie folgt vor:
   a. Erweitern Sie die Registerkarte und wählen Sie dann Verwenden einer vorhandenen Rolle aus.
   b. Wählen Sie die zuvor erstellte `LambdaS3Role` aus.
7. Wählen Sie „Event-Funktion erstellen“.

Um den Event-Funktionscode hochzuladen (Konsole)

1. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
2. Wählen Sie die ZIP-Datei aus.
3. Klicken Sie auf Upload.
4. Wählen Sie in der Dateiauswahl Ihre ZIP-Datei und anschließend Öffnen aus.
5. Wählen Sie Speichern.

AWS CLI

Um die Event-Funktion zu erstellen (AWS CLI)

- Führen Sie den CLI-Befehl für die von Ihnen gewählte Sprache aus. Stellen Sie sicher, dass Sie den `role` Parameter durch Ihre eigene AWS-Konto ID `123456789012` ersetzen. Ersetzen Sie für den Parameter `region` den Wert `us-east-1` durch die Region, in der Sie Ihre Amazon-S3-Buckets erstellt haben.

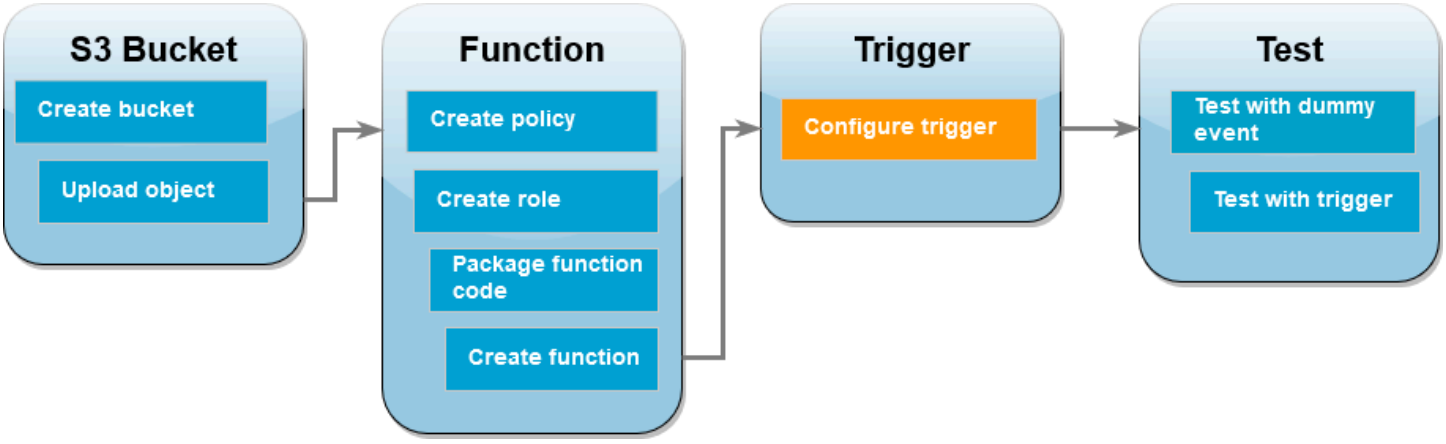
  - Führen Sie für Node.jsden folgenden Befehl in dem Verzeichnis aus, das Ihre `function.zip` Datei enthält.

    ```
    aws lambda create-function --function-name CreateThumbnail \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
    --timeout 10 --memory-size 1024 \
    --role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
    ```

  - Führen Sie für Python den folgenden Befehl in dem Verzeichnis aus, das die Datei `lambda_function.zip` enthält:

    ```
    aws lambda create-function --function-name CreateThumbnail \
    --zip-file fileb://lambda_function.zip --handler
     lambda_function.lambda_handler \
    --runtime python3.14 --timeout 10 --memory-size 1024 \
    --role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
    ```

Amazon S3 so konfigurieren, dass die Event-Funktion aufgerufen wird



Damit Ihre Lambda-Ereignisfunktion ausgeführt wird, wenn Sie ein Bild in Ihren Quell-Bucket hochladen, müssen Sie einen Trigger für Ihre Funktion konfigurieren. Sie können den Amazon-S3-Auslöser entweder über die Konsole oder mithilfe der AWS CLI konfigurieren.

> **Important**
>
> Mit diesem Verfahren wird der Amazon-S3-Bucket so konfiguriert, dass die Funktion jedes Mal aufgerufen wird, wenn ein Objekt im Bucket erstellt wird. Konfigurieren Sie dieses Verhalten nur für den Quell-Bucket. Wenn Ihre Lambda-Ereignisfunktion Objekte in demselben Bucket erstellt, in dem sie aufgerufen wird, kann Ihre Funktion kontinuierlich in einer Schleife aufgerufen werden. Dies kann dazu führen, dass Ihnen unerwartete Gebühren in Rechnung gestellt werden. AWS-Konto

AWS-Managementkonsole

So konfigurieren Sie den Amazon-S3-Auslöser (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion (`CreateThumbnail`) aus.
2. Wählen Sie Add trigger.
3. Wählen Sie S3 aus.
4. Wählen Sie unter Bucket Ihren Quell-Bucket aus.
5. Wählen Sie unter Ereignistypen die Option Alle Objekterstellungsereignisse aus.
6. Aktivieren Sie unter Rekursiver Aufruf das Kontrollkästchen, um zu bestätigen, dass die Verwendung desselben Amazon-S3-Buckets für die Ein- und Ausgabe nicht empfohlen wird. Weitere Informationen zu rekursiven Aufrufmustern in Lambda finden Sie bei Serverless Land unter Recursive patterns that cause run-away Lambda functions.
7. Wählen Sie Hinzufügen aus.

   Wenn Sie einen Auslöser über die Lambda-Konsole erstellen, erstellt Lambda automatisch eine ressourcenbasierte Richtlinie, um dem von Ihnen ausgewählten Dienst die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen.

AWS CLI

Um den Amazon S3-Trigger zu konfigurieren (AWS CLI)

1. Damit Ihr Amazon-S3-Quell-Bucket Ihre Funktion aufruft, wenn Sie eine Bilddatei hinzufügen, müssen Sie zunächst mithilfe einer ressourcenbasierten Richtlinie Berechtigungen für Ihre Funktion konfigurieren. Eine ressourcenbasierte Richtlinienerklärung erteilt anderen die AWS-Services Erlaubnis, Ihre Funktion aufzurufen. Führen Sie den folgenden CLI-Befehl aus, um Amazon S3 die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen. Achten Sie darauf, den `source-account` Parameter durch Ihre eigene AWS-Konto ID zu ersetzen und Ihren eigenen Quell-Bucket-Namen zu verwenden.

```
aws lambda add-permission --function-name CreateThumbnail \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-source-bucket \
--source-account 123456789012
```

   Die mit diesem Befehl definierte Richtlinie ermöglicht es Amazon S3 nur, Ihre Funktion aufzurufen, wenn eine Aktion in Ihrem Quell-Bucket ausgeführt wird.

   > **Note**
   >
   > Namen von Amazon-S3-Buckets sind zwar global eindeutig, trotzdem empfiehlt es sich, bei der Verwendung ressourcenbasierter Richtlinien anzugeben, dass der Bucket zu Ihrem Konto gehören muss. Denn wenn Sie einen Bucket löschen, kann ein anderer AWS-Konto Bucket einen Bucket mit demselben Amazon-Ressourcennamen (ARN) erstellen.

2. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `notification.json`: Wenn dieser JSON auf Ihren Quell-Bucket angewendet wird, konfiguriert er den Bucket so, dass jedes Mal, wenn ein neues Objekt hinzugefügt wird, eine Benachrichtigung an Ihre Lambda-Ereignisfunktion gesendet wird. Ersetzen Sie die AWS-Konto Nummer und den ARN AWS-Region in der Lambda Event Function durch Ihre eigene Kontonummer und Region.

```
{
"LambdaFunctionConfigurations": [
    {
```

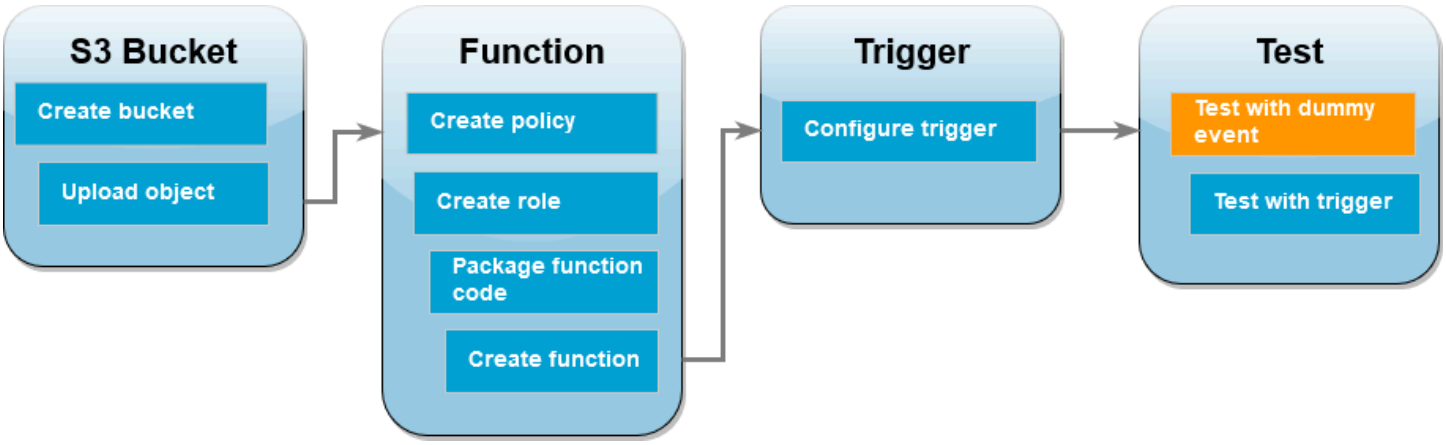
```
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Führen Sie den folgenden CLI-Befehl aus, um die Benachrichtigungseinstellungen in der von Ihnen erstellten JSON-Datei auf Ihren Quell-Bucket anzuwenden. Ersetzen Sie dabei `amzn-s3-demo-source-bucket` durch den Namen Ihres Quell-Buckets.

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-source-
bucket \
--notification-configuration file://notification.json
```

   Weitere Informationen zu dem `put-bucket-notification-configuration` Befehl und der `notification-configuration` Option finden Sie unter put-bucket-notification-configuration in der AWS CLI-Befehlsreferenz.

Testen Sie Ihre Lambda-Event-Funktion mit einem Dummy-Event



Bevor Sie Ihr gesamtes Setup testen, indem Sie Ihrem Amazon S3 S3-Quell-Bucket eine Image-Datei hinzufügen, testen Sie, ob Ihre Lambda-Ereignisfunktion ordnungsgemäß funktioniert, indem Sie sie mit einem Dummy-Ereignis aufrufen. Ein Ereignis in Lambda ist ein JSON-formatted Dokument, das Daten enthält, die Ihre Funktion verarbeiten kann. Wenn Ihre Funktion von Amazon S3 aufgerufen wird, enthält das an Ihre Funktion gesendete Ereignis Informationen wie den Bucket-Namen, den Bucket-ARN und den Objektschlüssel.

AWS-Managementkonsole

So testen Sie Ihre Lambda-Event-Funktion mit einem Dummy-Ereignis (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole und wählen Sie Ihre Funktion (`CreateThumbnail`) aus.
2. Wählen Sie die Registerkarte Test.
3. Gehen Sie im Bereich Testereignis wie folgt vor, um Ihr Testereignis zu erstellen:
   a. Wählen Sie unter Testereignisaktion die Option Neues Ereignis erstellen aus.
   b. Geben Sie für Event name (Ereignisname) **`myTestEvent`** ein.
   c. Wählen Sie unter Vorlage die Option S3 Put aus.
   d. Ersetzen Sie die Werte für die folgenden Parameter durch Ihre eigenen Werte:
      - Ersetzen Sie für durch den`awsRegion`, in `us-east-1` dem AWS-Region Sie Ihre Amazon S3 S3-Buckets erstellt haben.
      - Ersetzen Sie für `name` den Wert `amzn-s3-demo-bucket` durch den Namen Ihres eigenen Amazon-S3-Quell-Buckets.
      - Ersetzen Sie für `key` den Wert `test%2Fkey` durch den Dateinamen des Testobjekts, das Sie im Schritt Hochladen eines Testbilds in Ihren Quell-Bucket in Ihren Quell-Bucket hochgeladen haben.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
```

```
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

e. Wählen Sie Speichern.

4. Wählen Sie im Bereich Testereignis die Option Testen aus.
5. Gehen Sie wie folgt vor, um zu überprüfen, ob Ihre Funktion eine verkleinerte Version Ihres Bilds erstellt und in Ihrem Amazon-S3-Ziel-Bucket gespeichert hat:

   a. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole.

   b. Wählen Sie Ihren Ziel-Bucket aus und vergewissern Sie sich, dass die Datei, deren Größe angepasst wurde, im Bereich Objekte vorhanden ist.

AWS CLI

Um Ihre Lambda-Event-Funktion mit einem Dummy-Ereignis zu testen (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `dummyS3Event.json`: Ersetzen Sie die Werte für die folgenden Parameter durch Ihre eigenen Werte:

- Ersetzen Sie für durch den`awsRegion`, in `us-east-1` dem AWS-Region Sie Ihre Amazon S3 S3-Buckets erstellt haben.
- Ersetzen Sie für `name` den Wert `amzn-s3-demo-bucket` durch den Namen Ihres eigenen Amazon-S3-Quell-Buckets.
- Ersetzen Sie für `key` den Wert `test%2Fkey` durch den Dateinamen des Testobjekts, das Sie im Schritt Hochladen eines Testbilds in Ihren Quell-Bucket in Ihren Quell-Bucket hochgeladen haben.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
```

```
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

2. Rufen Sie die Funktion in dem Verzeichnis auf, in dem Sie die Datei `dummyS3Event.json` gespeichert haben, indem Sie den folgenden CLI-Befehl ausführen. Dieser Befehl ruft Ihre Lambda-Ereignisfunktion synchron auf, indem Sie sie `RequestResponse` als Wert für den Parameter invocation-type angeben. Weitere Informationen zu synchronen und asynchronen Aufrufen finden Sie unter Aufrufen von Lambda-Funktionen.

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

   Die Option „cli-binary-format“ ist erforderlich, wenn Sie die Version 2 von AWS CLI verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter AWS CLI -unterstützte globale Befehlszeilenoptionen.

3. Vergewissern Sie sich, dass Ihre Funktion eine Miniaturversion Ihres Bilds erstellt und in Ihrem Amazon-S3-Ziel-Bucket gespeichert hat. Führen Sie den folgenden CLI-Befehl aus und ersetzen Sie dabei `amzn-s3-demo-source-bucket-resized` durch den Namen Ihres eigenen Ziel-Buckets:

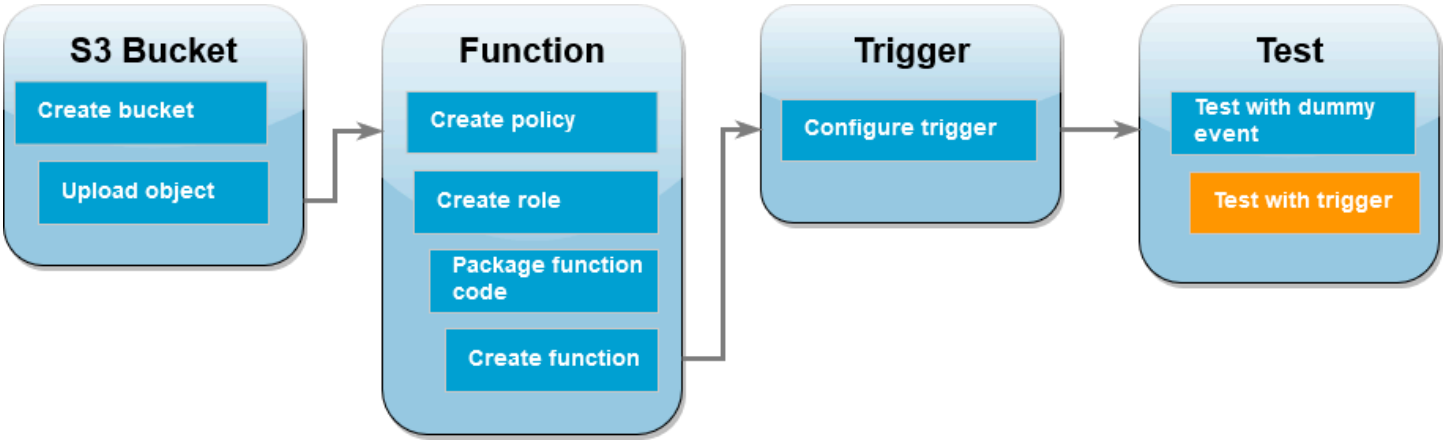
```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

   Die Ausgabe sollte in etwa wie folgt aussehen: Der Parameter `Key` zeigt den Dateinamen Ihrer verkleinerten Bilddatei an.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
```

```
            "StorageClass": "STANDARD"
        }
    ]
}
```

Testen Sie Ihre Event-Funktion mit dem Amazon S3 S3-Trigger



Nachdem Sie nun bestätigt haben, dass Ihre Lambda Event Function korrekt funktioniert, können Sie Ihr komplettes Setup testen, indem Sie Ihrem Amazon S3 S3-Quell-Bucket eine Image-Datei hinzufügen. Wenn Sie Ihr Bild zum Quell-Bucket hinzufügen, sollte Ihre Lambda-Ereignisfunktion automatisch aufgerufen werden. Ihre Funktion erstellt eine verkleinerte Version der Datei und speichert sie in Ihrem Ziel-Bucket.

AWS-Managementkonsole

So testen Sie Ihre Lambda-Ereignisfunktion mit dem Amazon S3 S3-Trigger (Konsole)

1. Gehen Sie wie folgt vor, um ein Bild in Ihren Amazon-S3-Bucket hochzuladen:
   a. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Quell-Bucket aus.
   b. Klicken Sie auf Upload.
   c. Wählen Sie Dateien hinzufügen und anschließend über die Dateiauswahl die Bilddatei aus, die Sie hochladen möchten. Ihr Bildobjekt kann eine beliebige JPG- oder PNG-Datei sein.
   d. Wählen Sie Öffnen und anschließend Hochladen aus.
2. Vergewissern Sie sich, dass Lambda eine verkleinerte Version Ihrer Bilddatei in Ihrem Ziel-Bucket gespeichert hat:

a. Navigieren Sie wieder zur Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Ziel-Bucket aus.

b. Im Bereich Objekte sollten Sie nun zwei Bilddateien mit geänderter Größe sehen, eine aus jedem Test Ihrer Lambda-Ereignisfunktion. Wählen Sie zum Herunterladen Ihres verkleinerten Bilds die Datei und anschließend Herunterladen aus.

AWS CLI

Um Ihre Lambda-Event-Funktion mit dem Amazon S3-Trigger zu testen (AWS CLI)

1. Führen Sie in dem Verzeichnis, das das hochzuladende Bild enthält, den folgenden CLI-Befehl aus. Ersetzen Sie dabei den Parameter `--bucket` durch den Namen Ihres Quell-Buckets. Verwenden Sie für die Parameter `--key` und `--body` den Dateinamen Ihres Testbilds. Ihr Testbild kann eine beliebige JPG- oder PNG-Datei sein.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key SmileyFace.jpg --
body ./SmileyFace.jpg
```

2. Vergewissern Sie sich, dass Ihre Funktion eine Miniaturversion Ihres Bilds erstellt und in Ihrem Amazon-S3-Ziel-Bucket gespeichert hat. Führen Sie den folgenden CLI-Befehl aus und ersetzen Sie dabei `amzn-s3-demo-source-bucket-resized` durch den Namen Ihres eigenen Ziel-Buckets:

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

Wenn Ihre Funktion erfolgreich ausgeführt wird, erhalten Sie eine Ausgabe, die der folgenden ähnelt. Ihr Ziel-Bucket sollte jetzt zwei verkleinerte Dateien enthalten.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
```

```
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die von Ihnen erstellte Richtlinie

1. Öffnen Sie die Seite Richtlinien in der IAM-Konsole.
2. Wählen Sie die Richtlinie aus, die Sie erstellt haben (AWSLambdaS3Policy).
3. Wählen Sie Policy actions (Richtlinienaktionen) und anschließend Delete (Löschen) aus.
4. Wählen Sie Löschen aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie den S3-Bucket:

1. Öffnen Sie die Amazon S3-Konsole.
2. Wählen Sie den Bucket aus, den Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen des Buckets in das Texteingabefeld ein.
5. Wählen Sie Bucket löschen aus.

Verwenden Sie Secrets Manager Manager-Geheimnisse in Lambda-Ereignisfunktionen

AWS Secrets Manager hilft Ihnen bei der Verwaltung von Anmeldeinformationen, API-Schlüsseln und anderen Geheimnissen, die Ihre Lambda-Ereignisfunktionen benötigen. Sie haben zwei Hauptansätze für das Abrufen von Geheimnissen in Ihren Lambda-Ereignisfunktionen, die beide eine bessere Leistung und geringere Kosten bieten als das direkte Abrufen von Geheimnissen mithilfe des SDK: AWS

- AWS Lambda-Erweiterung für Parameter und Geheimnisse — Eine laufzeitunabhängige Lösung, die eine einfache HTTP-Schnittstelle zum Abrufen von Geheimnissen bietet
- Powertools for AWS Lambda Parameters Utility — Eine codeintegrierte Lösung, die mehrere Anbieter (Secrets Manager, Parameter Store AppConfig) mit integrierten Transformationen unterstützt

Beide Ansätze verwenden lokale Caches für Secrets, sodass Ihre Funktion nicht bei jedem Aufruf den Secrets Manager aufrufen muss. Wenn Ihre Funktion ein Secret anfordert, wird zuerst der Cache überprüft. Ist das Secret verfügbar und nicht abgelaufen, wird es sofort zurückgegeben. Andernfalls wird es aus dem Secrets Manager abgerufen, zwischengespeichert und zurückgegeben. Dieser Caching-Mechanismus führt zu schnelleren Antwortzeiten und geringeren Kosten durch die Minimierung von API-Aufrufen.

Auswahl des Ansatzes

Berücksichtigen Sie diese Faktoren, wenn Sie zwischen der Erweiterung und folgenden Optionen wählen: PowerTools

Verwenden Sie die Lambda-Erweiterung AWS Parameters and Secrets, wenn:

- Sie eine laufzeitunabhängige Lösung benötigen, die mit jeder Lambda-Laufzeit funktioniert.
- Sie keine Codeabhängigkeiten zu Ihrer Funktion hinzufügen möchten.
- Sie nur Secrets aus Secrets Manager oder Parameter Store abrufen müssen.

Verwenden Sie Powertools for AWS Lambda Parameters Utility, wenn:

- Sie eine integrierte Entwicklungserfahrung mit Ihrem Anwendungscode wünschen.
- Sie benötigen Unterstützung für mehrere Anbieter (Secrets Manager, Parameter Store, AppConfig)
- Sie integrierte Datentransformationen (JSON-Parsing, Base64-Decodierung) benötigen.

- Sie verwenden Python- TypeScript, Java- oder .NET-Laufzeiten

Wann Sie den Secrets Manager mit Lambda verwenden sollten

Zu den häufigsten Anwendungsfällen für den Secrets Manager mit Lambda gehören:

- Speichern von Datenbank-Anmeldeinformationen, die Ihre Funktion für die Verbindung zu Amazon RDS oder anderen Datenbanken verwendet.
- Verwalten von API-Schlüsseln für externe Dienste, die Ihre Funktion aufruft.
- Speichern von Verschlüsselungsschlüsseln oder anderen sensiblen Konfigurationsdaten.
- Automatische Rotation der Anmeldeinformationen, ohne dass Sie Ihren Funktionscode aktualisieren müssen.

Verwendung der AWS Parameter und Geheimnisse Lambda-Erweiterung

Die Lambda-Erweiterung AWS Parameters and Secrets verwendet eine einfache HTTP-Schnittstelle, die mit jeder Lambda-Laufzeit kompatibel ist. Standardmäßig werden Secrets 300 Sekunden (5 Minuten) lang zwischengespeichert und es können bis zu 1 000 Secrets gespeichert werden. Sie können diese Einstellungen mit Umgebungsvariablen anpassen.

Verwenden Sie Secrets Manager in einer Lambda-Ereignisfunktion

In diesem Abschnitt wird davon ausgegangen, dass Sie bereits ein Secret im Secrets Manager besitzen. Informationen zum Erstellen eines Secrets finden Sie unter Create an AWS Secrets Manager Secret.

Erstellen des Bereitstellungspakets

Wählen Sie Ihre bevorzugte Laufzeit und folgen Sie den Schritten, um eine Funktion zu erstellen, die Secrets aus dem Secrets Manager abruft. Die Beispielfunktion ruft ein Secrets aus dem Secrets Manager ab und kann verwendet werden, um auf Datenbank-Anmeldeinformationen, API-Schlüssel oder andere sensible Konfigurationsdaten in Ihren Anwendungen zuzugreifen.

Python

So erstellen Sie eine Python-Event-Funktion

1. Erstellen Sie ein neues Projektverzeichnis und navigieren Sie dorthin. Beispiel:

```
mkdir my_function
cd my_function
```

2. Erstellen Sie eine Datei mit dem Namen `lambda_function.py` und dem folgenden Code. Verwenden Sie für `secret_name` den Namen oder den Amazon-Ressourcennamen (ARN) Ihres Secrets.

```
import json
import boto3
import requests

def lambda_handler(event, context):
    try:
        # Replace with the name or ARN of your secret
        secret_name = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"

        # Resolve the session token from the credential provider chain so that
 this
        # works across all initialization modes, including SnapStart. Resolve
 inside
        # the handler so the token stays fresh after a SnapStart restore.
        session_token =
 boto3.Session().get_credentials().get_frozen_credentials().token

        secrets_extension_endpoint = f"http://localhost:2773/secretsmanager/get?
secretId={secret_name}"
        headers = {"X-Aws-Parameters-Secrets-Token": session_token}

        response = requests.get(secrets_extension_endpoint, headers=headers)
        print(f"Response status code: {response.status_code}")

        secret = json.loads(response.text)["SecretString"]
        print(f"Retrieved secret: {secret}")

        return {
            'statusCode': response.status_code,
            'body': json.dumps({
                'message': 'Successfully retrieved secret',
                'secretRetrieved': True
            })
        }
```

```
    except Exception as e:
        print(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Error retrieving secret',
                'error': str(e)
            })
        }
```

3. Erstellen Sie eine Datei namens „`requirements.txt`“ mit folgendem Inhalt:

```
requests
```

4. Installieren Sie die Abhängigkeiten:

```
pip install -r requirements.txt -t .
```

5. Erstellen Sie eine ZIP-Datei, die alle Dateien enthält:

```
zip -r function.zip .
```

Node.js

Um eine Node.js Ereignisfunktion zu erstellen

1. Erstellen Sie ein neues Projektverzeichnis und navigieren Sie dorthin. Beispiel:

```
mkdir my_function
cd my_function
```

2. Erstellen Sie eine Datei mit dem Namen `index.mjs` und dem folgenden Code. Verwenden Sie für `secret_name` den Namen oder den Amazon-Ressourcennamen (ARN) Ihres Secrets.

```
import http from 'http';
import { fromNodeProviderChain } from '@aws-sdk/credential-providers';

export const handler = async (event) => {
    try {
```

```
        // Replace with the name or ARN of your secret
        const secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";

        // Resolve the session token from the credential provider chain so that
 this
        // works across all initialization modes, including SnapStart. Resolve
 inside
        // the handler so the token stays fresh after a SnapStart restore.
        const { sessionToken } = await fromNodeProviderChain()();

        const options = {
            hostname: 'localhost',
            port: 2773,
            path: `/secretsmanager/get?secretId=${secretName}`,
            headers: {
                'X-Aws-Parameters-Secrets-Token': sessionToken
            }
        };

        const response = await new Promise((resolve, reject) => {
            http.get(options, (res) => {
                let data = '';
                res.on('data', (chunk) => { data += chunk; });
                res.on('end', () => {
                    resolve({
                        statusCode: res.statusCode,
                        body: data
                    });
                });
            }).on('error', reject);
        });

        const secret = JSON.parse(response.body).SecretString;
        console.log('Retrieved secret:', secret);

        return {
            statusCode: response.statusCode,
            body: JSON.stringify({
                message: 'Successfully retrieved secret',
                secretRetrieved: true
            })
        };
    } catch (error) {
```

```
        console.error('Error:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'Error retrieving secret',
                error: error.message
            })
        };
    }
};
```

3. Erstellen Sie eine ZIP-Datei, die die Datei „`index.mjs`“ enthält:

```
zip -r function.zip index.mjs
```

Java

Um eine Java-Ereignisfunktion zu erstellen

1. Erstellen Sie ein Maven-Projekt:

```
mvn archetype:generate \
    -DgroupId=example \
    -DartifactId=lambda-secrets-demo \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DinteractiveMode=false
```

2. Navigieren Sie zum Projektverzeichnis:

```
cd lambda-secrets-demo
```

3. Öffnen Sie die Datei „`pom.xml`“ und ersetzen Sie deren Inhalt durch Folgendes:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>example</groupId>
    <artifactId>lambda-secrets-demo</artifactId>
```

```
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.1</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>auth</artifactId>
            <version>2.25.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

4. Benennen Sie die Datei „`/lambda-secrets-demo/src/main/java/example/App.java`“ in „`Hello.java`“ um, um sie an den Standardnamen des Java-Handlers von Lambda (`example.Hello::handleRequest`) anzupassen:

```
mv src/main/java/example/App.java src/main/java/example/Hello.java
```

5. Öffnen Sie die Datei `Hello.java` und ersetzen Sie den Inhalt durch Folgendes. Verwenden Sie für `secretName` den Namen oder den Amazon-Ressourcennamen (ARN) Ihres Secrets.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.auth.credentials.AwsSessionCredentials;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class Hello implements RequestHandler<Object, String> {
    private final HttpClient client = HttpClient.newHttpClient();
    private final DefaultCredentialsProvider credentialsProvider =
 DefaultCredentialsProvider.create();

    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Replace with the name or ARN of your secret
            String secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
            String endpoint = "http://localhost:2773/secretsmanager/get?
secretId=" + secretName;

            // Resolve the session token from the credential provider chain so
 that this
            // works across all initialization modes, including SnapStart.
 Resolve inside
            // the handler so the token stays fresh after a SnapStart restore.
            AwsSessionCredentials credentials = (AwsSessionCredentials)
 credentialsProvider.resolveCredentials();
            String sessionToken = credentials.sessionToken();
```

```
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(endpoint))
                .header("X-Aws-Parameters-Secrets-Token", sessionToken)
                .GET()
                .build();

            HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());

            String secret = response.body();
            secret = secret.substring(secret.indexOf("SecretString") + 15);
            secret = secret.substring(0, secret.indexOf("\""));

            System.out.println("Retrieved secret: " + secret);
            return String.format(
                "{\"statusCode\": %d, \"body\": \"%s\"}",
                response.statusCode(), "Successfully retrieved secret"
            );

        } catch (Exception e) {
            e.printStackTrace();
            return String.format(
                "{\"body\": \"Error retrieving secret: %s\"}",
                e.getMessage()
            );
        }
    }
}
```

6. Entfernen Sie das Testverzeichnis. Maven erstellt dies standardmäßig, aber wir benötigen es für dieses Beispiel nicht.

```
rm -rf src/test
```

7. Erstellen Sie das Projekt:

```
mvn package
```

8. Laden Sie die JAR-Datei (`target/function.jar`) zur späteren Verwendung herunter.

Erstellen Sie die Event-Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie für Function name (Funktionsname) **`secret-retrieval-demo`** ein.
5. Wählen Sie Ihre bevorzugte Laufzeit.
6. Wählen Sie „Event-Funktion erstellen“.

So laden Sie das Bereitstellungspaket hoch

1. Wählen Sie auf der Registerkarte Code der Funktion die Option Hochladen von und wählen Sie dann .zip-Datei (für Python und Node.js) oder .jar-Datei (für Java) aus.
2. Laden Sie das Bereitstellungspaket hoch, das Sie zuvor erstellt haben.
3. Wählen Sie Speichern.

Hinzufügen der Erweiterung

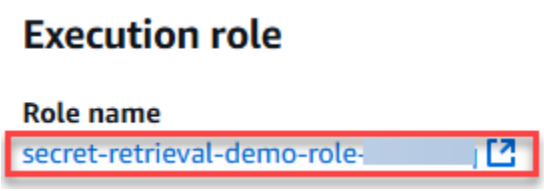
Um das hinzuzufügen AWS Parameter und Geheimnisse Lambda-Erweiterung als Ebene

1. Scrollen Sie auf der Registerkarte Code der Funktion nach unten zu Ebenen.
2. Wählen Sie Add a layer (Layer hinzufügen) aus.
3. Wählen Sie AWS -Ebenen aus.
4. Wählen Sie AWS- Parameters-and-Secrets-Lambda-Extension.
5. Wählen Sie die neueste Version aus.
6. Wählen Sie Hinzufügen aus.

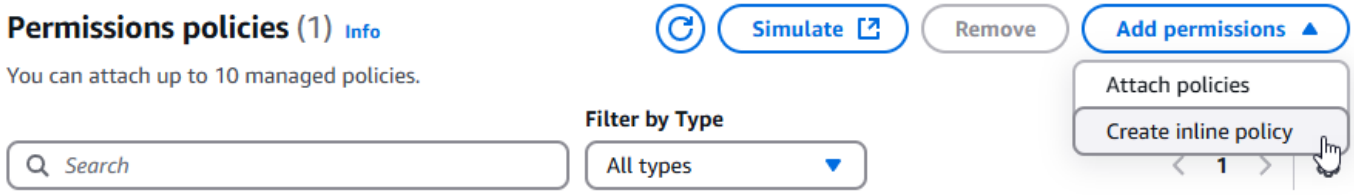
Berechtigungen hinzufügen

So fügen Sie Ihrer Ausführungsrolle Berechtigungen für den Secrets Manager hinzu

1. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
2. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



3. Wählen Sie Add permissions (Berechtigungen hinzufügen) und dann Create inline policy (Inline-Richtlinie erstellen) aus.



4. Wählen Sie die Registerkarte JSON aus und fügen Sie die folgende Richtlinie hinzu. Geben Sie für `Resource` den ARN Ihres Secrets ein.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"
        }
    ]
}
```

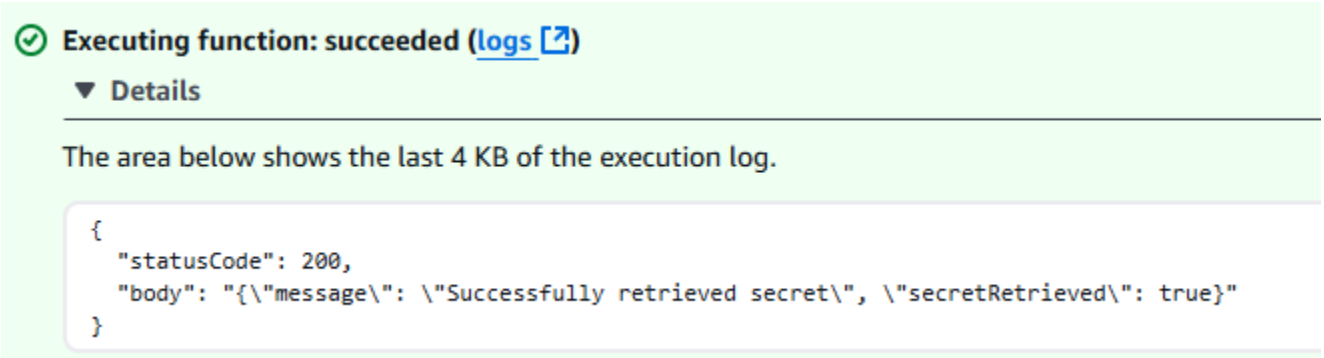
5. Wählen Sie Weiter aus.
6. Geben Sie einen Namen für die Richtlinie ein.
7. Wählen Sie Richtlinie erstellen aus.

Testen Sie die Event-Funktion

Um die Event-Funktion zu testen

1. Kehren Sie zur Lambda-Konsole zurück.

2. Wählen Sie die Registerkarte Test.
3. Wählen Sie Test aus. Sie sollten die folgende Antwort sehen:



Umgebungsvariablen

Die Lambda-Erweiterung AWS Parameters and Secrets verwendet die folgenden Standardeinstellungen. Sie können diese Einstellungen überschreiben, indem Sie die entsprechenden Umgebungsvariablen erstellen. Setzen Sie `PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL` auf `DEBUG`, um die aktuellen Einstellungen für eine Funktion anzuzeigen. Die Erweiterung CloudWatch protokolliert ihre Konfigurationsinformationen zu Beginn jedes Funktionsaufrufs in Logs.

Einstellung	Standardwert	Zulässige Werte	Umgebungsvariable	Details
HTTP-Port	2773	1–65 535	PARAMETERS_SECRETS_EXTENSION_HTTP_PORT	Port für den lokalen HTTP-Server
Cache aktiviert	TRUE	TRUE | FALSE	PARAMETERS_SECRETS_EXTENSION_CACHE_ENABLED	Aktivieren oder Deaktivieren des Cache
Cache-Größe	1000	0–1 000	PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE	Auf 0 setzen, um das Caching zu deaktivieren
Secrets Manager TTL	300 Sekunden	0–300 Sekunde	SECRETS_MANAGER_TTL	Time-to-live für zwischengespeicherte Geheimnisse. Auf

Einstellung	Standardwert	Zulässige Werte	Umgebungsvariable	Details
				0 setzen, um das Caching zu deaktivieren. Diese Variable wird ignoriert, wenn der Wert für PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE 0 ist.
Parameter Store TTL	300 Sekunden	0–300 Sekunde	SSM_PARAMETER_STORE_TTL	Time-to-live für zwischengespeicherte Parameter. Auf 0 setzen, um das Caching zu deaktivieren. Diese Variable wird ignoriert, wenn der Wert für PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE 0 ist.
Protokollebene	INFO	DEBUG | INFO | WARN | ERROR | NONE	PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL	Der Detaillierungsgrad, der in Protokollen für die Erweiterung gemeldet wird
Maximale Verbindungen	3	1 oder höher	PARAMETERS_SECRETS_EXTENSION_MAX_CONNECTIONS	Maximale Anzahl von HTTP-Verbindungen für Anforderungen an den Parameter Store oder Secrets Manager
Timeout für Secrets Manager	0 (kein Timeout)	Alle ganzen Zahlen	SECRETS_MANAGER_TIMEOUT_MILLIS	Timeout für Anfragen an Secrets Manager (in Millisekunden)

Einstellung	Standardwert	Zulässige Werte	Umgebungsvariable	Details
Timeout für Parameter Store	0 (kein Timeout)	Alle ganzen Zahlen	SSM_PARAMETER_STORE_TIMEOUT_MILLIS	Timeout für Anfragen an den Parameter Store (in Millisekunden)

Arbeiten mit Secret-Rotation

Wenn Sie Secrets häufig rotieren, kann die standardmäßige Cachedauer von 300 Sekunden dazu führen, dass Ihre Funktion veraltete Secrets verwendet. Sie haben zwei Möglichkeiten, um sicherzustellen, dass Ihre Funktion den neuesten Secret-Wert verwendet:

- Reduzieren Sie die Cache-TTL, indem Sie die `SECRETS_MANAGER_TTL`-Umgebungsvariable auf einen niedrigeren Wert (in Sekunden) setzen. Wenn Sie sie beispielsweise auf `60` einstellen, wird sichergestellt, dass Ihre Funktion niemals ein Secret verwendet, das älter als eine Minute ist.
- Verwenden Sie die Staging-Bezeichnungen `AWSCURRENT` oder `AWSPREVIOUS` in Ihrer Secret-Anfrage, um sicherzustellen, dass Sie die gewünschte Version erhalten:

```
secretsmanager/get?secretId=YOUR_SECRET_NAME&versionStage=AWSCURRENT
```

Wählen Sie den Ansatz, der Ihren Anforderungen an Leistung und Aktualität am besten gerecht wird. Eine niedrigere TTL bedeutet häufigere Aufrufe von Secrets Manager, stellt jedoch sicher, dass Sie mit den aktuellsten Secret-Werten arbeiten.

Verwenden Sie das Parameter-Hilfsprogramm von Powertools für AWS Lambda

Das Parameter-Hilfsprogramm von Powertools for AWS Lambda bietet eine einheitliche Oberfläche zum Abrufen von Geheimnissen von mehreren Anbietern, darunter Secrets Manager, Parameterspeicher und. AppConfig Es übernimmt Caching und Transformationen und bietet im Vergleich zum Erweiterungsansatz eine integriertere Entwicklungserfahrung.

Vorteile des Parameter-Dienstprogramms

- Mehrere Anbieter — Rufen Sie Parameter aus Secrets Manager und Parameter Store ab und AppConfig verwenden Sie dieselbe Schnittstelle

- Built-in Transformationen — Automatisches JSON-Parsing, Base64-Decodierung und andere Datentransformationen
- Integriertes Caching – Konfigurierbares Caching mit TTL-Unterstützung zur Reduzierung von API-Aufrufen
- Typsicherheit — Starke Eingabeunterstützung in und anderen unterstützten Laufzeiten TypeScript
- Fehlerbehandlung — Built-in Wiederholungslogik und Fehlerbehandlung

Codebeispiele

Die folgenden Beispiele zeigen, wie Secrets mit dem Parameter-Dienstprogramm in verschiedenen Laufzeiten abgerufen werden:

Python

> **Note**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm Powertools for AWS Lambda Parameters.

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities import parameters

logger = Logger()

def lambda_handler(event, context):
    try:
        # Get secret with caching (default TTL: 5 seconds)
        secret_value = parameters.get_secret("my-secret-name")

        # Get secret with custom TTL
        secret_with_ttl = parameters.get_secret("my-secret-name", max_age=300)

        # Get secret and transform JSON
        secret_json = parameters.get_secret("my-json-secret", transform="json")

        logger.info("Successfully retrieved secrets")
```

```
        return {
            'statusCode': 200,
            'body': 'Successfully retrieved secrets'
        }

    except Exception as e:
        logger.error(f"Error retrieving secret: {str(e)}")
        return {
            'statusCode': 500,
            'body': f'Error: {str(e)}'
        }
```

TypeScript

> **Note**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm Powertools for AWS Lambda Parameters.

```
import { Logger } from '@aws-lambda-powertools/logger';
import { getSecret } from '@aws-lambda-powertools/parameters/secrets';
import type { Context } from 'aws-lambda';

const logger = new Logger();

export const handler = async (event: any, context: Context) => {
    try {
        // Get secret with caching (default TTL: 5 seconds)
        const secretValue = await getSecret('my-secret-name');

        // Get secret with custom TTL
        const secretWithTtl = await getSecret('my-secret-name', { maxAge: 300 });

        // Get secret and transform JSON
        const secretJson = await getSecret('my-json-secret', { transform: 'json' });

        logger.info('Successfully retrieved secrets');
```

```
        return {
            statusCode: 200,
            body: 'Successfully retrieved secrets'
        };

    } catch (error) {
        logger.error('Error retrieving secret', { error });
        return {
            statusCode: 500,
            body: `Error: ${error}`
        };
    }
};
```

Java

> **Note**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm Powertools for AWS Lambda Parameters.

```
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.parameters.SecretsProvider;
import software.amazon.lambda.powertools.parameters.ParamManager;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class SecretHandler implements RequestHandler<Object, String> {

    private final SecretsProvider secretsProvider =
 ParamManager.getSecretsProvider();

    @Logging
    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Get secret with caching (default TTL: 5 seconds)
```

```
            String secretValue = secretsProvider.get("my-secret-name");

            // Get secret with custom TTL (300 seconds)
            String secretWithTtl = secretsProvider.withMaxAge(300).get("my-secret-
name");

            // Get secret and transform JSON
            MySecret secretJson = secretsProvider.get("my-json-secret",
 MySecret.class);

            return "Successfully retrieved secrets";

        } catch (Exception e) {
            return "Error retrieving secret: " + e.getMessage();
        }
    }

    public static class MySecret {
        // Define your secret structure here
    }
}
```

.NET

> **Note**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm Powertools for AWS Lambda Parameters.

```
using AWS.Lambda.Powertools.Logging;
using AWS.Lambda.Powertools.Parameters;
using Amazon.Lambda.Core;

[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

public class Function
{
```

```
    private readonly ISecretsProvider _secretsProvider;

    public Function()
    {
        _secretsProvider = ParametersManager.SecretsProvider;
    }

    [Logging]
    public async Task<string> FunctionHandler(object input, ILambdaContext context)
    {
        try
        {
            // Get secret with caching (default TTL: 5 seconds)
            var secretValue = await _secretsProvider.GetAsync("my-secret-name");

            // Get secret with custom TTL
            var secretWithTtl = await
 _secretsProvider.WithMaxAge(TimeSpan.FromMinutes(5))
                .GetAsync("my-secret-name");

            // Get secret and transform JSON
            var secretJson = await _secretsProvider.GetAsync<MySecret>("my-json-
secret");

            return "Successfully retrieved secrets";
        }
        catch (Exception e)
        {
            return $"Error retrieving secret: {e.Message}";
        }
    }

    public class MySecret
    {
        // Define your secret structure here
    }
}
```

Einrichtung und Berechtigungen

Um das Parameter-Dienstprogramm zu verwenden, müssen Sie Folgendes tun:

1. Installieren Sie Powertools für AWS Lambda für Ihre Runtime. Details hierzu finden Sie unter Elektrowerkzeuge für AWS Lambda.
2. Fügen Sie der Ausführungsrolle Ihrer Funktion die erforderlichen IAM-Berechtigungen hinzu. Weitere Einzelheiten finden Sie unter Verwaltung von Berechtigungen in AWS Lambda.
3. Konfigurieren Sie optionale Einstellungen über Umgebungsvariablen.

Die erforderlichen IAM-Berechtigungen sind dieselben wie beim Erweiterungsansatz. Das Hilfsprogramm verarbeitet automatisch Caching und API-Aufrufe an Secrets Manager basierend auf Ihrer Konfiguration.

Verwenden von Lambda mit Amazon SQS

> **Note**
>
> Wenn Sie Daten an ein anderes Ziel als eine Lambda-Ereignisfunktion senden oder die Daten vor dem Senden anreichern möchten, finden Sie weitere Informationen unter Amazon EventBridge Pipes.

Sie können eine Lambda-Ereignisfunktion verwenden, um Nachrichten in einer Amazon Simple Queue Service (Amazon SQS) -Warteschlange zu verarbeiten. Lambda unterstützt sowohl Standard-Warteschlangen als auch First-In-First-Out-Warteschlangen (FIFO) für Zuordnungen von Ereignisquellen. Sie können den Bereitstellungsmodus auch verwenden, um spezielle Abfrageressourcen für Ihre Amazon SQS SQS-Ereignisquellenzuordnungen zuzuweisen. Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden AWS-Region, obwohl sie sich unterscheiden können. AWS-Konten

Eine Beispielfunktion, die Amazon SQS SQS-Nachrichten verarbeitet, finden Sie unterthe section called “Tutorial”. Informationen zur kontenübergreifenden Verarbeitung finden Sie unter. the section called “SQS-Kontoübergreifendes Tutorial” Eine vollständige Liste der Konfigurationsoptionen, finden Sie unter the section called “Parameters”.

Bei der Verarbeitung von Amazon-SQS-Nachrichten müssen Sie eine Logik für teilweise Batch-Antworten implementieren, um zu verhindern, dass versucht wird, erfolgreich verarbeitete Nachrichten erneut zu verarbeiten, wenn einige Nachrichten in einem Batch fehlschlagen. Das Batch Processor Utility von Powertools for AWS Lambda vereinfacht diese Implementierung, indem es automatisch die Logik der partiellen Batch-Antwort verarbeitet, wodurch die Entwicklungszeit reduziert und die Zuverlässigkeit verbessert wird.

Themen

- Verständnis des Polling- und Batching-Verhaltens für Amazon-SQS-Zuordnungen von Ereignisquellen
- Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen
- Konfiguration des Bereitstellungsmodus für die Zuordnung von Amazon SQS SQS-Ereignisquellen
- Beispiel für ein Standard-Warteschlangen-Nachrichtenereignis
- Beispiel für ein FIFO-Warteschlangen-Nachrichtenereignis
- Erstellen und Konfigurieren einer Amazon SQS-Zuordnung von Ereignisquellen

- Konfiguration des Skalierungsverhaltens für SQS-Zuordnungen von Ereignisquellen
- Behandlung von Fehlern für eine SQS-Ereignisquelle in Lambda
- Lambdaparameter für Amazon-SQS-Zuordnungen von Ereignisquellen
- Verwendung der Ereignisfilterung mit einer Amazon-SQS-Ereignisquelle
- Tutorial: Verwenden von Lambda mit Amazon SQS
- Tutorial: Verwenden einer kontoübergreifenden Amazon SQS SQS-Warteschlange als Ereignisquelle

Verständnis des Polling- und Batching-Verhaltens für Amazon-SQS-Zuordnungen von Ereignisquellen

Mit Amazon-SQS-Zuordnungen von Ereignisquellen fragt Lambda die Warteschlange ab und ruft Ihre Funktion synchron mit einem Ereignis auf. Jedes Ereignis kann einen Batch von mehreren Nachrichten aus der Warteschlange enthalten. Lambda empfängt diese Ereignisse jeweils stapelweise und ruft Ihre Funktion für jeden Stapel einmal auf. Wenn Ihre Funktion einen Batch erfolgreich verarbeitet, löscht Lambda deren Nachrichten aus der Warteschlange.

Wenn Lambda einen Batch empfängt, bleiben die Nachrichten in der Warteschlange, werden aber für die Dauer der Zeitbeschränkung für die Sichtbarkeit der Warteschlange ausgeblendet. Wenn Ihre Funktion alle Nachrichten im Batch erfolgreich verarbeitet hat, löscht Lambda die Nachrichten aus der Warteschlange. Wenn Ihre Funktion bei der Verarbeitung eines Batches auf einen Fehler stößt, werden standardmäßig alle Nachrichten in diesem Batch nach Ablauf des Zeitlimits für die Sichtbarkeit wieder in der Warteschlange sichtbar. Deshalb muss der Funktionscode in der Lage sein, dieselbe Nachricht mehrmals ohne unbeabsichtigte Begleiterscheinungen zu verarbeiten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Um zu verhindern, dass Lambda eine Nachricht mehrfach verarbeitet, können Sie entweder Ihre Ereignisquellenzuordnung so konfigurieren, dass Batch-Elementfehler in Ihre Funktionsantwort

aufgenommen werden, oder Sie können die DeleteMessageAPI verwenden, um Nachrichten aus der Warteschlange zu entfernen, wenn Ihre Lambda-Ereignisfunktion sie erfolgreich verarbeitet.

Weitere Informationen zu Konfigurationsparametern, die Lambda für SQS-Zuordnungen von Ereignisquellen unterstützt, finden Sie unter the section called “Erstellen einer SQS-Zuordnung von Ereignisquellen”.

Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen

Für Workloads, bei denen Sie den Durchsatz Ihrer Zuordnung von Ereignisquellen optimieren müssen, können Sie den Bereitstellungsmodus verwenden. Im Bereitstellungsmodus definieren Sie Mindest- und Höchstgrenzen für die Anzahl der bereitgestellten Ereignis-Poller. Diese bereitgestellten Event-Poller sind auf Ihre Zuordnung von Ereignisquellen ausgerichtet und können unerwartete Nachrichtenspitzen durch reaktionsschnelle Autoskalierung bewältigen. Die mit dem Bereitstellungsmodus konfigurierte Amazon SQS SQS-Ereignisquellenzuweisung skaliert dreimal schneller (bis zu 1.000 gleichzeitige Aufrufe pro Minute) und unterstützt eine 80-mal höhere Parallelität (bis zu 100.000 gleichzeitige Aufrufe) als die standardmäßige Amazon SQS SQS-Funktion zur Zuordnung von Ereignisquellen.

Wir empfehlen, den Bereitstellungsmodus für ereignisgesteuerte Amazon SQS SQS-Workloads zu verwenden, für die strenge Leistungsanforderungen gelten, z. B. Finanzdienstleister, die Marktdatenfeeds verarbeiten, E-Commerce-Plattformen, die personalisierte Empfehlungen in Echtzeit bereitstellen, und Spieleunternehmen, die Live-Spielerinteraktionen verwalten. Die Verwendung des Bereitstellungsmodus verursacht zusätzliche Kosten. Eine detaillierte Preisgestaltung finden Sie unter Preise.AWS Lambda

Jeder Event Poller im Bereitstellungsmodus kann bis zu 1 MB/s Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten. Der Bereich der akzeptierten Werte für die Mindestanzahl von Event-Pollern (MinimumPollers) liegt zwischen 2 und 200, der Standardwert ist 2. Der Bereich der akzeptierten Werte für die maximale Anzahl von Event-Pollern (MaximumPollers) liegt zwischen 2 und 10.000, der Standardwert ist 200. MaximumPollers muss größer oder gleich sein MinimumPollers.

Ermittlung der erforderlichen Event-Poller

Um die Anzahl der Ereignisabfragen zu schätzen, die erforderlich sind, um eine optimale Nachrichtenverarbeitungsleistung sicherzustellen, wenn Sie den Bereitstellungsmodus für SQS ESM verwenden, erfassen Sie die folgenden Metriken für Ihre Anwendung: maximale SQS-Ereignisse pro

Sekunde, die eine Verarbeitung mit niedriger Latenz erfordern, durchschnittliche Größe der SQS-Ereignisnutzlast, durchschnittliche Dauer der Lambda-Ereignisfunktion und konfigurierte Batchgröße.

Zunächst können Sie anhand der folgenden Formel die Anzahl der SQS-Ereignisse pro Sekunde (EPS) schätzen, die von einem Event Poller für Ihren Workload unterstützt werden:

```
EPS per event poller =
        minimum(
            ceiling(1024 / average event size in KB),
            ceiling(10 / average function duration in seconds) * batch size,
            min(100, 10 * batch size)
                )
```

Anschließend können Sie anhand der folgenden Formel die Anzahl der mindestens erforderlichen Poller berechnen. Diese Berechnung stellt sicher, dass Sie ausreichend Kapazität bereitstellen, um Ihren Spitzenverkehrsanforderungen gerecht zu werden.

```
Required event pollers = (Peak number of events per second in Queue) / EPS per event
 poller
```

Stellen Sie sich einen Workload mit einer Standard-Batchgröße von 10, einer durchschnittlichen Ereignisgröße von 3 KB, einer durchschnittlichen Funktionsdauer von 100 ms und der Anforderung, 1.000 Ereignisse pro Sekunde zu verarbeiten, vor. In diesem Szenario unterstützt jeder Event Poller ungefähr 100 Ereignisse pro Sekunde (EPS). Daher empfehlen wir, die Mindestanzahl an Pollern auf 10 festzulegen, um Ihren Anforderungen an den Spitzenverkehr angemessen gerecht zu werden. Wenn Ihr Workload dieselben Merkmale aufweist, aber eine durchschnittliche Funktionsdauer von 1 Sekunde aufweist, unterstützt jeder Poller nur 10 EPS. Sie müssen also mindestens 100 Poller konfigurieren, um 1.000 Ereignisse pro Sekunde bei niedriger Latenz zu unterstützen.

Wir empfehlen die Verwendung einer Standard-Batchgröße von 10 oder höher, um die Effizienz von Event-Pollern im Bereitstellungsmodus zu maximieren. Höhere Batchgrößen ermöglichen es jedem Poller, mehr Ereignisse pro Aufruf zu verarbeiten, was den Durchsatz und die Kosteneffizienz verbessert. Berücksichtigen Sie bei der Planung Ihrer Event-Poller-Kapazität potenzielle Verkehrsspitzen und ziehen Sie in Betracht, den MinimumPollers-Wert etwas höher als das berechnete Minimum festzulegen, um einen Puffer bereitzustellen. Überwachen Sie außerdem Ihre Workload-Merkmale im Zeitverlauf, da Änderungen der Nachrichtengröße, der Funktionsdauer oder der Verkehrsmuster möglicherweise Anpassungen an Ihrer Event-Poller-Konfiguration erforderlich machen, um eine optimale Leistung und Kosteneffizienz zu gewährleisten. Für eine genaue

Kapazitätsplanung empfehlen wir, Ihre spezifische Arbeitslast zu testen, um den tatsächlichen Gewinn pro Leistung zu ermitteln, den jeder Event Poller erzielen kann.

Konfiguration des Bereitstellungsmodus für die Zuordnung von Amazon SQS SQS-Ereignisquellen

Sie können den Bereitstellungsmodus für Ihre Amazon SQS SQS-Ereignisquellenzuordnung mithilfe der Konsole oder der Lambda-API konfigurieren.

So konfigurieren Sie den Bereitstellungsmodus für eine bestehende Amazon SQS SQS-Ereignisquellenzuordnung (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion mit der Amazon SQS SQS-Ereignisquellenzuordnung aus, für die Sie den Bereitstellungsmodus konfigurieren möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie die Amazon SQS SQS-Ereignisquellenzuordnung aus, für die Sie den Bereitstellungsmodus konfigurieren möchten, und klicken Sie dann auf Bearbeiten.
5. Wählen Sie unter Konfiguration der Zuordnung von Ereignisquellen die Option Bereitstellungsmodus konfigurieren aus.
   - Geben Sie für Minimum Event Pollers einen Wert zwischen 2 und 200 ein. Wenn Sie keinen Wert angeben, wählt Lambda den Standardwert 2.
   - Geben Sie für Maximum Event Pollers einen Wert zwischen 2 und 10.000 ein. Dieser Wert muss größer oder gleich Ihrem Wert für Minimum Event Pollers sein. Wenn Sie keinen Wert angeben, wählt Lambda einen Standardwert von 200.
6. Wählen Sie Speichern.

Sie können den Bereitstellungsmodus programmgesteuert mithilfe des `ProvisionedPollerConfig` Objekts in Ihrem konfigurieren. `EventSourceMappingConfiguration` Der folgende `UpdateEventSourceMapping` CLI-Befehl konfiguriert beispielsweise einen `MinimumPollers` Wert von 5 und einen `MaximumPollers` Wert von 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

Nachdem Sie den Bereitstellungsmodus konfiguriert haben, können Sie die Verwendung von Event-Pollern für Ihren Workload beobachten, indem Sie die `ProvisionedPollers`-Metrik überwachen. Weitere Informationen finden Sie unter Metriken zur Zuordnung von Ereignisquellen.

Um den Bereitstellungsmodus zu deaktivieren und zum Standardmodus (auf Abruf) zurückzukehren, können Sie den folgenden `UpdateEventSourceMapping` CLI-Befehl verwenden:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

> **Note**
>
> Der Bereitstellungsmodus kann nicht in Verbindung mit der Einstellung für maximale Parallelität verwendet werden. Wenn Sie den Bereitstellungsmodus verwenden, steuern Sie die maximale Parallelität durch die maximale Anzahl von Event-Pollern.

Weitere Informationen zur Konfiguration des Bereitstellungsmodus finden Sie unter. the section called “Zuordnung erstellen”

Beispiel für ein Standard-Warteschlangen-Nachrichtenereignis

Example Amazon-SQS-Nachrichtenereignis (Standardwarteschlange)

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {
                "myAttribute": {
                    "stringValue": "myValue",
```

```
                    "stringListValues": [],
                    "binaryListValues": [],
                    "dataType": "String"
                }
            },
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        }
    ]
}
```

Lambda fragt standardmäßig bis zu 10 Nachrichten in Ihrer Warteschlange sofort ab und sendet diesen Batch an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle so konfigurieren, dass die Datensätze bis zu 5 Minuten lang gepuffert werden, indem Sie ein Batch-Fenster konfigurieren. Vor dem Aufrufen der Funktion fragt Lambda weiter Nachrichten aus der Standardwarteschlange ab, bis das Batch-Fenster abläuft, das Nutzlastkontingent pro Aufruf erreicht ist oder die konfigurierte maximale Batch-Größe erreicht ist.

Wenn Sie ein Batch-Fenster verwenden und Ihre SQS-Warteschlange sehr wenig Datenverkehr enthält, wartet Lambda möglicherweise bis zu 20 Sekunden, bevor Sie Ihre Funktion aufruft. Dies gilt auch, wenn Sie ein Batch-Fenster unter 20 Sekunden festlegen.

> **Note**
>
> In Java können beim Deserialisieren von JSON Nullzeigerfehler auftreten. Dies könnte daran liegen, dass die Groß- und Kleinschreibung von „Records“ und „eventSourceARN“ vom JSON-Objektmapper konvertiert wird.

Beispiel für ein FIFO-Warteschlangen-Nachrichtenereignis

Bei FIFO-Warteschlangen enthalten Datensätze zusätzliche Attribute, die mit der Deduplizierung und Sequenzierung zusammenhängen.

Example Amazon-SQS-Nachrichtenereignis (FIFO-Warteschlange)

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

Erstellen und Konfigurieren einer Amazon SQS-Zuordnung von Ereignisquellen

Um Amazon-SQS-Nachrichten mit Lambda zu verarbeiten, konfigurieren Sie Ihre Warteschlange mit den entsprechenden Einstellungen und erstellen Sie dann eine Lambda-Zuordnung von Ereignisquellen.

Themen

- Konfigurieren einer Warteschlange zur Verwendung mit Lambda
- Einrichten von Lambda-Ausführungsrollenberechtigungen
- Erstellen einer SQS-Zuordnung von Ereignisquellen

Konfigurieren einer Warteschlange zur Verwendung mit Lambda

Wenn Sie noch keine Amazon SQS SQS-Warteschlange haben, erstellen Sie eine, die als Ereignisquelle für Ihre Lambda-Ereignisfunktion dient. Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden AWS-Region, obwohl sie sich unterscheiden können. AWS-Konten

Damit Ihre Funktion genügend Zeit hat, um jeden Stapel von Datensätzen zu verarbeiten, setzen Sie die Zeitüberschreitung für die Sichtbarkeit der Ausgangswarteschlange auf mindestens das Sechsfache der Konfigurationszeitüberschreitung Ihrer Funktion. Diese zusätzliche Zeit ermöglicht es für Lambda, einen erneuten Versuch zu machen, wenn die Funktion gedrosselt wird, während ein früherer Batch verarbeitet wird.

> **Note**
>
> Das Timeout Ihrer Funktion muss kleiner oder gleich dem Sichtbarkeits-Timeout der Warteschlange sein. Lambda validiert diese Anforderung, wenn Sie eine Ereignisquellenzuordnung erstellen oder aktualisieren, und gibt einen Fehler zurück, wenn das Funktions-Timeout das Sichtbarkeits-Timeout der Warteschlange überschreitet.

Wenn Lambda bei der Verarbeitung eines Stapels auf einen Fehler stößt, kehren standardmäßig alle Nachrichten in diesem Stapel in die Warteschlange zurück. Nach dem Sichtbarkeits-Timeout werden die Nachrichten wieder für Lambda sichtbar. Sie können Ihre Zuordnung von Ereignisquellen so konfigurieren, dass Sie partielle Batch-Antworten verwenden, um nur die fehlgeschlagenen Nachrichten an die Warteschlange zurückzugeben.

Wenn Ihre Funktion eine Nachricht mehrmals nicht verarbeiten kann, kann Amazon SQS sie außerdem an eine Warteschlange für unzustellbare Nachrichten senden. Wir empfehlen, das `maxReceiveCount` in der Redrive-Richtlinie Ihrer Quellwarteschlange auf mindestens 5 zu setzen. Dadurch erhält Lambda einige Chancen, es erneut zu versuchen, bevor fehlgeschlagene Nachrichten direkt an die Warteschlange für unzustellbare Nachrichten gesendet werden.

Einrichten von Lambda-Ausführungsrollenberechtigungen

Die AWSLambdaSQSQueueExecutionRole AWS verwaltete Richtlinie umfasst die Berechtigungen, die Lambda benötigt, um aus Ihrer Amazon SQS SQS-Warteschlange zu lesen. Sie können diese verwaltete Richtlinie der Ausführungsrolle Ihrer Funktion hinzufügen.

Wenn Sie optional eine verschlüsselte Warteschlange verwenden, müssen Sie Ihrer Ausführungsrolle auch die folgende Berechtigung hinzufügen:

- kms:Decrypt

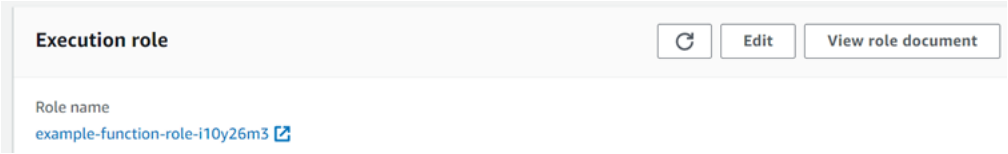
Erstellen einer SQS-Zuordnung von Ereignisquellen

Erstellen Sie eine Ereignisquellenzuordnung, um Lambda anzuweisen, Elemente aus Ihrer Warteschlange an eine Lambda-Ereignisfunktion zu senden. Sie können mehrere Ereignisquellen-Zuweisungen zum Verarbeiten von Elementen aus mehreren Warteschlangen mit nur einer Funktion erstellen. Wenn Lambda die Zielfunktion aufruft, kann das Ereignis mehrere Elemente bis zu einer konfigurierbaren maximalen Batch-Größe enthalten.

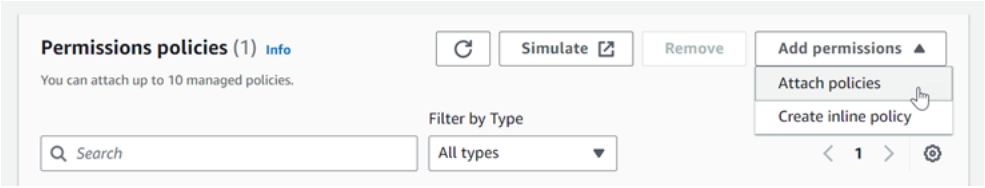
Um Ihre Funktion für das Lesen aus Amazon SQS zu konfigurieren, fügen Sie die AWSLambdaSQSQueueExecutionRole AWS verwaltete Richtlinie Ihrer Ausführungsrolle hinzu. Erstellen Sie anschließend mithilfe der folgenden Schritte eine Zuordnung der SQS-Ereignisquellen von der Konsole aus.

So fügen Sie Berechtigungen hinzu und erstellen einen Auslöser

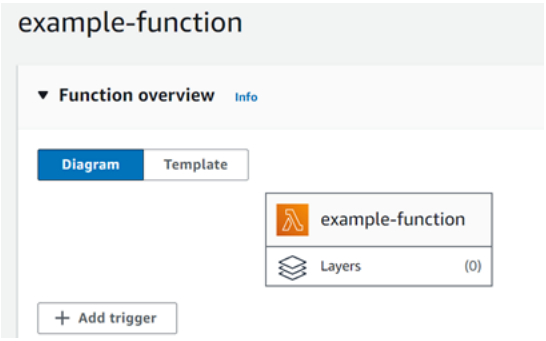
1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



5. Wählen Sie Berechtigungen hinzufügen aus und wählen Sie dann Richtlinien direkt anhängen aus.



6. Geben Sie im Suchfeld `AWSLambdaSQSQueueExecutionRole` ein. Fügen Sie diese Richtlinie zu Ihrer Ausführungsrolle hinzu. Dies ist eine AWS verwaltete Richtlinie, die die Berechtigungen enthält, die Ihre Funktion zum Lesen aus einer Amazon SQS SQS-Warteschlange benötigt. Weitere Informationen zu dieser Richtlinie finden Sie AWSLambdaSQSQueueExecutionRolein der Referenz zu AWS verwalteten Richtlinien.
7. Gehen Sie zurück zu Ihrer Funktion in der Lambda-Konsole. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).



8. Wählen Sie einen Auslösertyp aus.
9. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Konfigurationsoptionen für Amazon-SQS-Ereignisquellen:

SQS Queue

Die Amazon-SQS-Warteschlange, aus der Datensätze gelesen werden sollen. Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden AWS-Region, obwohl sie sich unterscheiden können. AWS-Konten

Aktivieren von Auslösern

Der Status der Zuordnung von Ereignisquellen. Enable trigger (Auslöser aktivieren) ist standardmäßig ausgewählt.

Batch-Größe

Die maximale Anzahl der Datensätze, die in jedem Stapel an die Funktion gesendet werden sollen. Bei einer Standardwarteschlange können dies bis zu 10.000 Datensätze sein. Bei einer FIFO-Warteschlange liegt der Höchstwert bei 10. Bei einer Stapelgröße über 10 müssen Sie das Stapelfenster (`MaximumBatchingWindowInSeconds`) zusätzlich auf mindestens 1 Sekunde festlegen.

Konfigurieren Sie Ihr Funktions-Timeout so, dass genügend Zeit bleibt, um einen ganzen Stapel von Elementen zu verarbeiten. Wenn die Verarbeitung bestimmter Elemente lange Zeit in Anspruch nimmt, wählen Sie eine kleinere Batch-Größe. Eine große Batchgröße kann die Effizienz für Workloads erhöhen, die sehr schnell sind oder viel Overhead aufweisen. Wenn Sie reservierte Nebenläufigkeit für Ihre Funktion konfigurieren, legen Sie mindestens fünf gleichzeitige Ausführungen fest, um das Risiko von Drosselungsfehlern zu senken, wenn Lambda die Funktion aufruft.

Lambda übergibt alle Datensätze im Batch in einem einzigen Aufruf an die Funktion, solange die Gesamtgröße der Ereignisse nicht das Nutzlastkontingent pro Aufruf für synchrone Aufrufe überschreitet (6 MB). Sowohl Lambda als auch Amazon SQS generieren Metadaten für jeden Datensatz. Diese zusätzlichen Metadaten werden auf die Gesamtnutzlastgröße angerechnet und können dazu führen, dass die Gesamtzahl der in einem Batch gesendeten Datensätze niedriger ist als die konfigurierte Batch-Größe. Die Metadatenfelder, die Amazon SQS sendet, können in der Länge variabel sein. Weitere Informationen zu den Amazon SQS-Metadatenfeldern finden Sie in der Dokumentation zum ReceiveMessageAPI-Betrieb in der Amazon Simple Queue Service API-Referenz.

Batchfenster

Die maximale Zeitspanne zur Erfassung von Datensätzen vor dem Aufruf der Funktion in Sekunden. Dies gilt nur für Standardwarteschlangen.

Wenn Sie ein Stapelfenster von mehr als 0 Sekunden verwenden, müssen Sie die erhöhte Verarbeitungszeit im Sichtbarkeits-Timeout Ihrer Warteschlange berücksichtigen. Wir empfehlen, das Sichtbarkeits-Timeout Ihrer Warteschlange auf das Sechsfache Ihres Timeouts für Event-Funktionen zuzüglich des Werts von festzulegen. `MaximumBatchingWindowInSeconds`

Dadurch hat Ihre Lambda-Ereignisfunktion Zeit, jeden Ereignisstapel zu verarbeiten und es im Falle eines Drosselungsfehlers erneut zu versuchen.

Wenn Nachrichten verfügbar werden, beginnt Lambda, Nachrichten stapelweise zu verarbeiten. Lambda beginnt mit der gleichzeitigen Verarbeitung von fünf Stapeln mit fünf gleichzeitigen Aufrufen Ihrer Funktion. Wenn Nachrichten noch verfügbar sind, addiert Lambda bis zu 300 gleichzeitige Aufrufe Ihrer Funktion pro Minute, bis zu einem Maximum von 1.250 gleichzeitigen Aufrufen.

Im Bereitstellungsmodus kann jeder Event Poller bis zu 1 MB/s Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten. Lambda skaliert die Anzahl der Event-Poller zwischen Ihrem konfigurierten Minimum und Maximum und fügt schnell bis zu 1.000 gleichzeitige Aufrufe pro Minute hinzu, um eine Verarbeitung Ihrer Amazon SQS SQS-Ereignisse mit geringer Latenz zu gewährleisten. Sie steuern Skalierung und Parallelität über diese minimalen und maximalen Event-Poller-Einstellungen. Weitere Informationen zur Skalierung und Parallelität von Event-Funktionen finden Sie unter. Die Skalierung von Lambda-Ereignisfunktionen verstehen

Um mehr Nachrichten zu verarbeiten, können Sie Ihre Lambda Event Function für einen höheren Durchsatz optimieren. Weitere Informationen finden Sie unter Grundlegendes zur AWS Lambda Skalierung mit Amazon SQS SQS-Standardwarteschlangen.

Filterkriterien

Verwenden Sie Filterkriterien, um zu steuern, welche Ereignisse Lambda zur Verarbeitung an Ihre Funktion sendet. Weitere Informationen finden Sie unter Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet.

Maximale Parallelität

Die maximale Anzahl der gleichzeitigen Funktionen, die die Ereignisquelle aufrufen kann. Kann nicht verwendet werden, wenn der Bereitstellungsmodus aktiviert ist. Weitere Informationen finden Sie unter Konfigurieren der maximalen Gleichzeitigkeit für Amazon-SQS-Ereignisquellen.

Modus bereitgestellter Kapazität

Wenn diese Option aktiviert ist, werden spezielle Abfrageressourcen für Ihre Ereignisquellenzuordnung zugewiesen. Sie können die minimale (2-200) und die maximale (2-2000) Anzahl von Event-Pollern konfigurieren. Jeder Event Poller kann bis zu 1 MB/sec Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten.

> **Note**
>
> Hinweis: Sie können den Bereitstellungsmodus und die Option Maximale Parallelität nicht zusammen verwenden. Wenn der Bereitstellungsmodus aktiviert ist, verwenden Sie die Einstellung für maximale Polleranzahl, um die Parallelität zu steuern.

Konfiguration des Skalierungsverhaltens für SQS-Zuordnungen von Ereignisquellen

Sie können das Skalierungsverhalten Ihrer Amazon SQS SQS-Ereignisquellenzuordnungen entweder über die Einstellungen für maximale Parallelität oder durch die Aktivierung des Bereitstellungsmodus steuern. Diese Optionen schließen sich gegenseitig aus.

Standardmäßig skaliert Lambda Ereignis-Poller automatisch auf der Grundlage des Nachrichtenvolumens. Wenn Sie den Bereitstellungsmodus aktivieren, weisen Sie eine Mindest- und Höchstzahl von dedizierten Abfrageressourcen zu, die bereit sind, die erwarteten Datenverkehrsmuster zu verarbeiten. Sie können die Leistung Ihrer Ereignisquellenzuordnung auf zwei Arten optimieren:

- Standardmodus (Standard): Lambda verwaltet die Skalierung automatisch, beginnend mit einer kleinen Anzahl von Pollern und dann je nach Arbeitslast nach oben oder unten.
- Bereitgestellter Modus: Sie konfigurieren dedizierte Polling-Ressourcen mit Mindest- und Höchstgrenzen, was eine dreimal schnellere Skalierung und eine bis zu 16-mal höhere Verarbeitungskapazität ermöglicht.

Bei Standardwarteschlangen verwendet Lambda Langabfrage, um eine Warteschlange so lange abzufragen, bis sie aktiv wird. Wenn Nachrichten verfügbar sind, beginnt Lambda mit der Verarbeitung von fünf Batches gleichzeitig mit fünf gleichzeitigen Aufrufen Ihrer Event-Funktion. Wenn Nachrichten weiterhin verfügbar sind, erhöht Lambda die Anzahl der Prozesse, die Batches lesen, um bis zu 300 gleichzeitige Aufrufe pro Minute. Die maximale Anzahl von Aufrufen, die eine Ereignisquellenzuordnung gleichzeitig verarbeiten kann, beträgt 1.250. Wenn der Verkehr gering ist, reduziert Lambda die Verarbeitung auf fünf gleichzeitige Aufrufe und kann auf bis zu zwei gleichzeitige Aufrufe optimiert werden, um die Amazon SQS SQS-Aufrufe und die entsprechenden Kosten zu reduzieren. Diese Optimierung ist jedoch nicht verfügbar, wenn Sie die Einstellung für maximale Gleichzeitigkeit aktivieren.

Bei FIFO-Warteschlangen sendet Lambda Nachrichten in der Reihenfolge, in der sie empfangen wurden, an Ihre Funktion. Wenn Sie eine Nachricht an eine FIFO-Warteschlange senden, geben Sie

eine Nachrichtengruppen-ID an. Amazon SQS stellt sicher, dass Nachrichten in derselben Gruppe in der Reihenfolge an Lambda übermittelt werden. Wenn Lambda Ihre Nachrichten in Batches liest, kann jeder Batch Nachrichten aus mehr als einer Nachrichtengruppe enthalten, aber die Reihenfolge der Nachrichten wird beibehalten. Wenn die Funktion einen Fehler zurückgibt, führt sie alle Wiederholungsversuche für die betroffenen Nachrichten aus, bevor Lambda weitere Nachrichten von derselben Gruppe erhält.

Im Bereitstellungsmodus kann jeder Event Poller bis zu 1 MB/sec Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten. Lambda skaliert die Anzahl der Event-Poller zwischen Ihrem konfigurierten Minimum und Maximum und fügt schnell bis zu 1.000 Parallelität pro Minute hinzu, um eine konsistente Verarbeitung Ihrer Amazon SQS SQS-Ereignisse mit niedriger Latenz zu gewährleisten. Die Verwendung des Bereitstellungsmodus verursacht zusätzliche Kosten. Eine detaillierte Preisgestaltung finden Sie unter Preise.AWS Lambda

Jeder Event Poller verwendet lange Abfragen in Ihrer SQS-Warteschlange mit bis zu 10 Abfragen pro Sekunde, wodurch Kosten für SQS-API-Anfragen anfallen. Einzelheiten finden Sie unter Amazon SQS SQS-Preise. Sie steuern Skalierung und Parallelität über diese minimalen und maximalen Event-Poller-Einstellungen, anstatt die Einstellung für maximale Parallelität zu verwenden, da diese Optionen nicht zusammen verwendet werden können.

> **Note**
>
> Sie können die Einstellung für maximale Parallelität und den Bereitstellungsmodus nicht gleichzeitig verwenden. Wenn der Bereitstellungsmodus aktiviert ist, steuern Sie die Skalierung und Parallelität Ihrer Amazon SQS SQS-Ereignisquellenzuordnung durch die minimale und maximale Anzahl von Event-Pollern.

Konfigurieren der maximalen Gleichzeitigkeit für Amazon-SQS-Ereignisquellen

Sie können die Einstellung für die maximale Gleichzeitigkeit verwenden, um das Skalierungsverhalten für Ihre SQS-Ereignisquellen zu steuern. Beachten Sie, dass maximale Parallelität nicht verwendet werden kann, wenn der Bereitstellungsmodus aktiviert ist. Die Einstellung für die maximale Gleichzeitigkeit begrenzt die Anzahl der gleichzeitigen Instances der Funktion, die eine Amazon-SQS-Ereignisquelle aufrufen kann. Die maximale Gleichzeitigkeit ist eine Einstellung auf der Ebene der Ereignisquelle. Wenn Sie einer Funktion mehrere Amazon-SQS-Ereignisquellen zugeordnet haben, kann jede Ereignisquelle eine separate Einstellung für die maximale Gleichzeitigkeit haben.

Sie können die maximale Parallelität verwenden, um zu verhindern, dass eine Warteschlange die gesamte reservierte Gleichzeitigkeit der Funktion oder den Rest des Gleichzeitigkeitskontingents des Kontos verwendet. Für die Konfiguration der maximalen Gleichzeitigkeit auf einer Amazon-SQS-Ereignisquelle fallen keine Gebühren an.

Wichtig ist, dass maximale und reservierte Parallelität zwei unabhängige Einstellungen sind. Setzen Sie die maximale Parallelität nicht höher als die reservierte Parallelität der Funktion. Wenn Sie maximale Parallelität konfigurieren, stellen Sie sicher, dass die reservierte Parallelität Ihrer Funktion größer oder gleich der gesamten maximalen Parallelität für alle Amazon SQS-Ereignisquellen auf der Funktion ist. Andernfalls könnte Lambda Ihre Nachrichten drosseln.

Wenn die Gleichzeitigkeitsquote Ihres Kontos auf den Standardwert von 1 000 eingestellt ist, kann eine Amazon-SQS-Zuordnung von Ereignisquellen skaliert werden, um Funktionsinstanzen bis zu diesem Wert aufzurufen, es sei denn, Sie geben eine maximale Gleichzeitigkeit an.

Wenn Sie eine Erhöhung der standardmäßigen Parallelitätsquote Ihres Kontos erhalten, ist Lambda möglicherweise nicht in der Lage, Instanzen für gleichzeitige Funktionen bis zu Ihrem neuen Kontingent aufzurufen. Standardmäßig kann Lambda so skaliert werden, dass bis zu 1.250 gleichzeitige Funktionsinstanzen für eine Amazon-SQS-Zuordnung von Ereignisquellen aufgerufen werden. Wenn Sie eine höhere Parallelität benötigen, aktivieren Sie den Bereitstellungsmodus. Der Bereitstellungsmodus skaliert auf bis zu 100.000 gleichzeitige Funktionsaufrufen. Weitere Informationen finden Sie unter Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen.

> **Note**
>
> Bei FIFO-Warteschlangen sind gleichzeitige Aufrufe entweder durch die Anzahl der Nachrichtengruppen-IDs (`messageGroupId`) oder die maximale Gleichzeitigkeitseinstellung begrenzt, je nachdem, welcher Wert niedriger ist. Wenn Sie beispielsweise sechs Nachrichtengruppen-IDs haben und die maximale Gleichzeitigkeit auf 10 festgelegt ist, kann Ihre Funktion maximal sechs gleichzeitige Aufrufe haben.

Sie können die maximale Gleichzeitigkeit für neue und vorhandene Zuordnung von Ereignisquellen in Amazon SQS konfigurieren.

Konfigurieren der maximalen Gleichzeitigkeit mithilfe der Lambda-Konsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option SQS aus. Dadurch wird die Registerkarte Configuration (Konfiguration) geöffnet.
4. Wählen Sie den Amazon-SQS-Auslöser aus und klicken Sie auf Edit (Bearbeiten).
5. Geben Sie für Maximum concurrency (Maximale Gleichzeitigkeit) eine Zahl zwischen 2 und 1 000 ein. Um die maximale Gleichzeitigkeit zu deaktivieren, lassen Sie das Feld leer.
6. Wählen Sie Speichern.

Konfigurieren Sie die maximale Parallelität mit dem AWS Command Line Interface (AWS CLI)

Verwenden Sie den update-event-source-mapping-Befehl mit der Option `--scaling-config`. Beispiel:

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

Um die maximale Gleichzeitigkeit zu deaktivieren, geben Sie einen leeren Wert für `--scaling-config` ein:

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

Konfigurieren der maximalen Gleichzeitigkeit mithilfe der Lambda-API

Verwenden Sie die UpdateEventSourceMappingAktion CreateEventSourceMappingoder mit einem ScalingConfigObjekt.

Behandlung von Fehlern für eine SQS-Ereignisquelle in Lambda

Um Fehler im Zusammenhang mit einer SQS-Ereignisquelle zu behandeln, verwendet Lambda automatisch eine Wiederholungsstrategie mit einer Backoff-Strategie. Sie können das Verhalten bei der Fehlerbehandlung auch anpassen, indem Sie Ihre SQS-Zuordnung von Ereignisquellen so konfigurieren, dass sie partielle Batch-Antworten zurückgibt.

Backoff-Strategie für fehlgeschlagene Aufrufe

Wenn ein Aufruf fehlschlägt, versucht Lambda, den Aufruf zu wiederholen, während eine Backoff-Strategie implementiert wird. Die Backoff-Strategie unterscheidet sich geringfügig, je nachdem, ob Lambda den Fehler aufgrund eines Fehlers in Ihrem Funktionscode oder aufgrund einer Drosselung festgestellt hat.

- Wenn Ihr Eventfunktionscode den Fehler verursacht hat, stoppt Lambda die Verarbeitung und versucht erneut, den Aufruf auszuführen. In der Zwischenzeit zieht sich Lambda allmählich zurück und reduziert die Menge der Gleichzeitigkeit, die Ihrer Amazon SQS-Zuordnung von Ereignisquellen zugewiesen wird. Wenn das Sichtbarkeits-Timeout Ihrer Warteschlange abgelaufen ist, wird die Nachricht wieder in der Warteschlange angezeigt.
- Wenn der Aufruf aufgrund von Drosselung fehlschlägt, setzt Lambda schrittweise Wiederholungsversuche zurück, indem es die Menge der Gleichzeitigkeit reduziert, die Ihrer Zuordnung von Ereignisquellen in Amazon SQS zugewiesen ist. Lambda fährt fort, die Nachricht erneut zu versuchen, bis der Zeitstempel der Nachricht das Sichtbarkeits-Timeout Ihrer Warteschlange überschreitet, an welchem Punkt Lambda die Nachricht verwirft.

Implementierung von partiellen Batch-Antworten

Wenn Ihre Lambda-Ereignisfunktion bei der Verarbeitung eines Batches auf einen Fehler stößt, werden alle Nachrichten in diesem Batch standardmäßig wieder in der Warteschlange sichtbar, einschließlich Nachrichten, die Lambda erfolgreich verarbeitet hat. Infolgedessen kann es passieren, dass die Funktion dieselbe Nachricht mehrmals verarbeitet.

Damit nicht erfolgreich bearbeiten Nachrichten in einem fehlgeschlagenen Batch erneut verarbeitet werden, können Sie die Zuordnung von Ereignisquellen so konfigurieren, dass nur die fehlgeschlagenen Nachrichten wieder sichtbar werden. Dies wird als partielle Batch-Antwort bezeichnet. Um partielle Batch-Antworten zu aktivieren, geben Sie `ReportBatchItemFailures` bei der Konfiguration Ihrer Ereignisquellenzuordnung die entsprechende FunctionResponseTypesAktion an. Mit dieser Einstellung kann Ihre Funktion einen teilweisen Erfolg zurückgeben, was dazu beitragen kann, die Anzahl unnötiger Wiederholungsversuche bei Datensätzen zu reduzieren.

> **Note**
>
> Das Batch Processor Utility von Powertools for AWS Lambda verarbeitet die gesamte partielle Batch-Antwortlogik automatisch. Dieses Tool vereinfacht die Implementierung von

Batch-Verarbeitungsmustern und reduziert den Bedarf an benutzerdefiniertem Code, der für die korrekte Behandlung von Fehlern bei Batch-Elementen erforderlich ist. Es ist für Python, Java, Typescript und .NET verfügbar.

Wenn `ReportBatchItemFailures` aktiviert ist, skaliert Lambda die Nachrichtenabfrage nicht herunter, wenn Funktionsaufrufe fehlschlagen. Wenn Sie erwarten, dass einige Nachrichten ausfallen – und Sie nicht möchten, dass sich diese Fehler auf die Nachrichtenverarbeitungsrate auswirken – verwenden Sie `ReportBatchItemFailures`.

Note

Berücksichtigen Sie bei Verwendung von partiellen Batch-Antworten Folgendes:

- Wenn die Funktion eine Ausnahme ausgibt, gilt der gesamte Batch als fehlgeschlagen.
- Wenn Sie dieses Feature mit einer FIFO-Warteschlange verwenden, sollte Ihre Funktion die Verarbeitung von Nachrichten nach dem ersten Fehler beenden und alle fehlgeschlagenen und nicht verarbeiteten Nachrichten in `batchItemFailures` zurückgeben. Das hilft, die Reihenfolge der Nachrichten in der Warteschlange beizubehalten.

So aktivieren Sie partielle Batchberichterstattung

1. Sehen Sie sich die bewährten Methoden für die Implementierung von partiellen Batch-Antworten an.
2. Führen Sie den folgenden Befehl aus, um `ReportBatchItemFailures` für Ihre Funktion zu aktivieren. Um die UUID Ihrer Ereignisquellenzuordnung abzurufen, führen Sie den Befehl list-event-source-mappings aus. AWS CLI

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. Aktualisieren Sie Ihren Funktionscode, um alle Ausnahmen abzufangen und fehlgeschlagene Nachrichten in einer `batchItemFailures`-JSON-Antwort zurückzugeben. Die `batchItemFailures`-Antwort muss eine Liste von Nachrichten-IDs als `itemIdentifier`-JSON-Werte enthalten.

Angenommen, Sie haben einen Batch von fünf Nachrichten mit den Nachrichten-IDs `id1`, `id2`, `id3`, `id4` und `id5`. Die Funktion verarbeitet erfolgreich `id1`, `id3` und `id5`. Damit die Nachrichten `id2` und `id4` in der Warteschlange wieder sichtbar werden, sollte Ihre Funktion folgende Antwort zurückgeben:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

Hier sind einige Beispiele für Funktionscodes, die die Liste der fehlgeschlagenen Nachrichten-IDs im Batch zurückgeben:

.NET

SDK für .NET

> **Note**
>
> Es GitHub gibt noch mehr dazu. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
```

```
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
```

```
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
 {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();

        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
```

```
                //process your message
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
 list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batch-Elementfehlern mit Lambda unter Verwendung von. JavaScript

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
```

```
    console.log(`Processed message: ${record.body}`);
}
```

Melden von SQS-Batch-Elementfehlern mit Lambda unter Verwendung von. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
```

```
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}
```

```
        for record in event["Records"]:
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
```

```
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
```

```
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Wenn die fehlgeschlagenen Ereignisse nicht in die Warteschlange zurückkehren, finden Sie weitere Informationen unter Wie behebe ich Probleme mit der Lambda-Funktion ReportBatchItemFailures SQS? im AWS Knowledge Center.

Erfolgs- und Misserfolgsbedingungen

Lambda behandelt einen Batch als komplett erfolgreich, wenn die Funktion eines der folgenden Elemente zurückgibt:

- Eine leere `batchItemFailures`-Liste
- Eine ungültige `batchItemFailures`-Liste
- Ein leeres `EventResponse`
- Ein ungültiges `EventResponse`

Lambda behandelt einen Batch als komplett fehlgeschlagen, wenn die Funktion eines der folgenden Elemente zurückgibt:

- Eine ungültige JSON-Antwort
- Eine leere Zeichenfolge `itemIdentifier`
- Ein ungültiges `itemIdentifier`
- Ein `itemIdentifier` mit einem falschen Schlüsselnamen
- Einen `itemIdentifier`-Wert mit einer Nachrichten-ID, die nicht existiert

CloudWatch Metriken

Um festzustellen, ob Ihre Funktion Fehler bei Chargenartikeln korrekt meldet, können Sie die Amazon SQS-Metriken `NumberOfMessagesDeleted` und die `ApproximateAgeOfOldestMessage` Amazon SQS-Metriken in Amazon CloudWatch überwachen.

- `NumberOfMessagesDeleted` verfolgt die Anzahl der Nachrichten, die aus der Warteschlange entfernt wurden. Wenn der Wert auf 0 sinkt, ist dies ein Zeichen dafür, dass die Funktionsantwort fehlgeschlagene Nachrichten nicht korrekt zurückgibt.
- `ApproximateAgeOfOldestMessage` verfolgt, wie lange die älteste Nachricht in der Warteschlange geblieben ist. Ein starker Anstieg dieser Metrik kann darauf hinweisen, dass die Funktion fehlgeschlagene Nachrichten nicht korrekt zurückgibt.

Verwenden von Powertools für AWS Lambda Batch-Prozessor

Das Batchverarbeitungsprogramm von Powertools for verarbeitet AWS Lambda automatisch die Logik für partielle Batch-Antworten und reduziert so die Komplexität der Implementierung von Batch-Fehlerberichten. Hier sind Beispiele für die Verwendung des Batch-Prozessors:

Python

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Amazon SQS SQS-Nachrichten mit AWS Lambda Batch-Prozessor.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import SQSEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.SQS)
logger = Logger()

def record_handler(record):
```

```
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Amazon SQS SQS-Nachrichten mit AWS Lambda Batch-Prozessor.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { SQSEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.SQS);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: SQSEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Lambdaparameter für Amazon-SQS-Zuordnungen von Ereignisquellen

Alle Lambda-Ereignisquelltypen verwenden dieselben CreateEventSourceMappingUpdateEventSourceMappingAPI-Operationen. Allerdings gelten nur einige der Parameter für Amazon SQS.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	10	Bei Standardw arteschlangen beträgt der Maximalwert 10 000. Bei FIFO-Warteschlange n beträgt der Maximalwert 10.
Aktiviert	N	true	Keine
EventSourceArn	Y	N/A	Der ARN des Datenstroms oder eines Stream-Ko nsumenten
FunctionName	Y	N/A	Keine
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Event-Funktion sendet
FunctionResponseTy pes	N	N/A	Damit Ihre Event-Funktion bestimmte Fehler in einem Batch melden kann, fügen Sie den Wert `ReportBat chItemFailures` in `FunctionR`

Parameter	Erforderlich	Standard	Hinweise
			esponseTypes ein. Weitere Informationen finden Sie unter Implementierung von partiellen Batch-Antworten.
MaximumBatchingWindowInSeconds	N	0	Das Batching-Fenster wird für FIFO-Warteschlangen nicht unterstützt
ProvisionedPollerConfig	N	N/A	Konfiguriert die minimale (2-200) und maximale (2-2000) Anzahl von dedizierten Event-Pollern für die SQS-Ereignisquellenzuordnung. Jeder Poller kann bis zu 1 MB/sec Durchsatz und 10 gleichzeitige Aufrufe verarbeiten.
ScalingConfig	N	N/A	Konfigurieren der maximalen Gleichzeitigkeit für Amazon-SQS-Ereignisquellen

Verwendung der Ereignisfilterung mit einer Amazon-SQS-Ereignisquelle

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Event-Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für Amazon-SQS-Ereignisquellen.

> **Note**
>
> Amazon-SQS-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `body`-Schlüssel.

Themen

- Grundlagen der Amazon-SQS-Ereignisfilterung

Grundlagen der Amazon-SQS-Ereignisfilterung

Angenommen, Ihre Amazon-SQS-Warteschlange enthält Nachrichten im folgenden JSON-Format.

```
{
    "RecordNumber": 1234,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

Ein Beispieldatensatz für diese Warteschlange würde wie folgt aussehen.

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 1234,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

Um anhand des Inhalts Ihrer Amazon-SQS-Nachrichten zu filtern, verwenden Sie den `body`-Schlüssel im Amazon-SQS-Nachrichtendatensatz. Angenommen, Sie möchten nur die Datensätze verarbeiten, bei denen `RequestCode` in Ihrer Amazon SQS SQS-Nachricht „BBBB“ steht. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer AWS SAM Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

Angenommen, Sie möchten, dass Ihre Event-Funktion nur die Datensätze verarbeitet, deren Wert größer als `RecordNumber` 9999 ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

Sie können Ihren Filter mithilfe der Konsole AWS CLI oder einer AWS SAM Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

Führen Sie den folgenden Befehl aus, um mithilfe von AWS Command Line Interface (AWS CLI) eine neue Ereignisquellenzuordnung mit diesen Filterkriterien zu erstellen.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

Um diesen Filter mithilfe hinzuzufügen AWS SAM, fügen Sie der YAML-Vorlage für Ihre Ereignisquelle den folgenden Ausschnitt hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

Für Amazon SQS kann der Nachrichtenkörper eine beliebige Zeichenfolge sein. Dies kann jedoch problematisch sein, wenn die `FilterCriteria` erwarten, dass `body` ein gültiges JSON-Format

hat. Umgekehrt gilt dasselbe – wenn der Textkörper der eingehenden Nachricht ein JSON-Format aufweist, die Filterkriterien jedoch erwarten, dass `body` eine einfache Zeichenfolge ist, kann dies zu unbeabsichtigtem Verhalten führen.

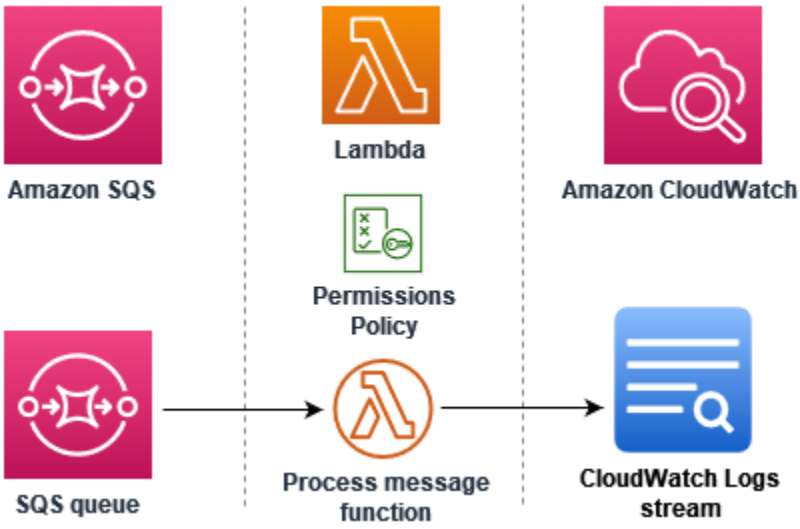
Um dieses Problem zu vermeiden, stellen Sie sicher, dass das Körperformat in Ihren `FilterCriteria` mit dem erwarteten Format von `body` in den Nachrichten übereinstimmt, die Sie von Ihrer Warteschlange erhalten. Bevor Sie Ihre Nachrichten filtern, wertet Lambda automatisch das Format des Körpers der eingehenden Nachricht und Ihres Filtermusters für `body` aus. Wenn es keine Übereinstimmung gibt, verwirft Lambda die Nachricht. In der folgenden Tabelle ist diese Auswertung zusammengefasst:

body-Format der eingehenden Nachricht	**body**-Format des Filtermusters	Resultierende Aktion
Einfache Zeichenfolge	Einfache Zeichenfolge	Lambda filtert basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Gültiges JSON	Lambda verwirft die Nachricht.
Gültiges JSON	Einfache Zeichenfolge	Lambda verwirft die Nachricht.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.

Tutorial: Verwenden von Lambda mit Amazon SQS

In diesem Tutorial erstellen Sie eine Lambda-Ereignisfunktion, die Nachrichten aus einer Amazon Simple Queue Service (Amazon SQS) -Warteschlange verarbeitet. Die Lambda-Ereignisfunktion wird

immer dann ausgeführt, wenn der Warteschlange eine neue Nachricht hinzugefügt wird. Die Funktion schreibt die Nachrichten in einen Amazon CloudWatch Logs-Stream. Das folgende Diagramm zeigt die AWS -Ressourcen, die Sie zur Durchführung des Tutorials verwenden.



Führen Sie für dieses Tutorial die folgenden Schritte aus:

1. Erstellen Sie eine Lambda-Ereignisfunktion, die Nachrichten in CloudWatch Logs schreibt.
2. Erstellen einer Amazon SQS-Warteschlange
3. Erstellen Sie eine Zuordnung von Ereignisquellen in Lambda. Die Zuordnung der Ereignisquelle liest die Amazon SQS SQS-Warteschlange und ruft Ihre Lambda-Ereignisfunktion auf, wenn eine neue Nachricht hinzugefügt wird.
4. Testen Sie das Setup, indem Sie Nachrichten zu Ihrer Warteschlange hinzufügen und die Ergebnisse in Logs überwachen. CloudWatch

Voraussetzungen

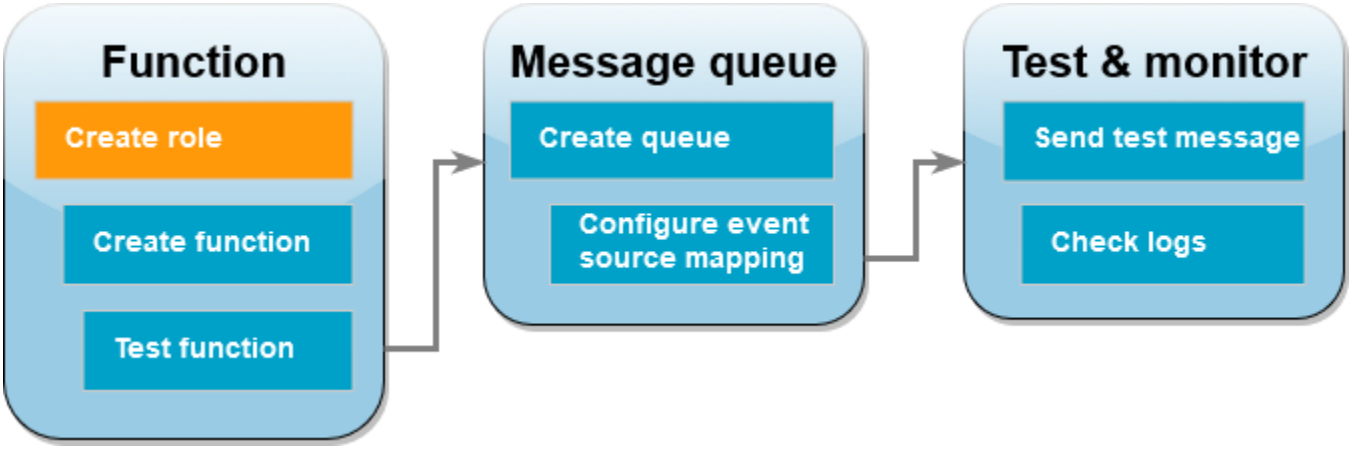
Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle



Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Ereignisfunktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. Damit Ihre Funktion Elemente aus Amazon SQS lesen kann, fügen Sie die AWSLambdaSQSQueueExecutionRoleBerechtigungsrichtlinie bei.

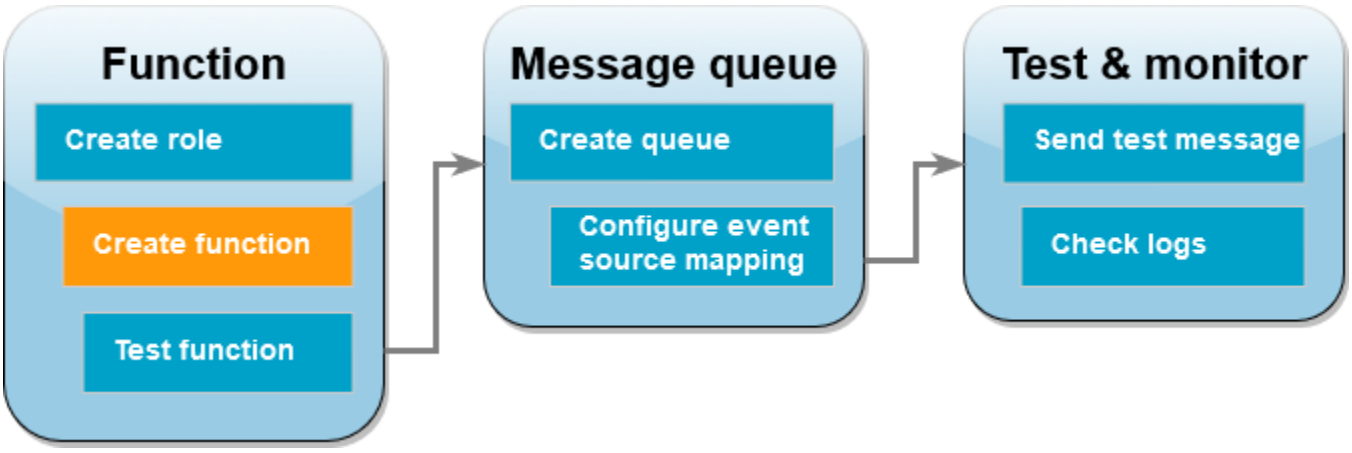
So erstellen Sie eine Ausführungsrolle und fügen eine Amazon-SQS-Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall die Option Lambda aus.
5. Wählen Sie Weiter aus.
6. Geben Sie im Suchfeld Berechtigungsrichtlinien die Zeichenfolge **`AWSLambdaSQSQueueExecutionRole`** ein.
7. Wählen Sie die AWSLambdaSQSQueueExecutionRoleRichtlinie aus und klicken Sie dann auf Weiter.

8. Geben Sie unter Rollendetails für Rollenname den Namen **`lambda-sqs-role`** ein und wählen Sie anschließend Rolle erstellen aus.

Schreiben Sie sich nach der Erstellung der Rolle den Amazon-Ressourcennamen (ARN) Ihrer Ausführungsrolle auf. Sie werden ihn in späteren Schritten noch benötigen.

Erstellen Sie die Event-Funktion



Erstellen Sie eine Lambda-Ereignisfunktion, die Ihre Amazon SQS SQS-Nachrichten verarbeitet. Der Funktionscode protokolliert den Hauptteil der Amazon SQS-Nachricht in CloudWatch Logs.

Dieses Tutorial verwendet die Node.js 24-Runtime, aber wir haben auch Beispielcode in anderen Runtime-Sprachen bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um Code für die gewünschte Laufzeit anzusehen. Der JavaScript Code, den Sie in diesem Schritt verwenden, befindet sich im ersten Beispiel, das auf der JavaScriptRegisterkarte angezeigt wird.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines SQS-Ereignisses mit Lambda unter Verwendung. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Konsumieren eines SQS-Ereignisses mit Lambda unter Verwendung. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

So erstellen Sie eine Node.js Lambda-Ereignisfunktion

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir sqs-tutorial
cd sqs-tutorial
```

2. Kopieren Sie den JavaScript Beispielcode in eine neue Datei mit dem Namen`index.js`.
3. Erstellen Sie ein Bereitstellungspaket mit dem folgenden `zip`-Befehl.

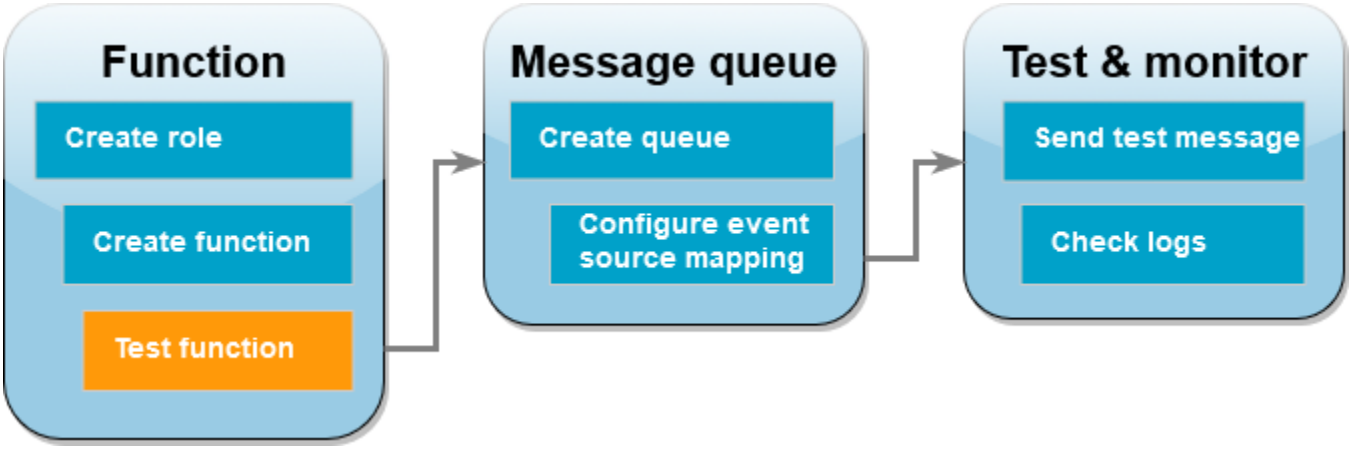
```
zip function.zip index.js
```

4. Erstellen Sie eine Lambda-Ereignisfunktion mit dem Befehl AWS CLI create-function. Geben Sie für den `role`-Parameter den ARN der Ausführungsrolle ein, die Sie zuvor erstellt haben.

> **Note**
>
> Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden. AWS-Region

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

Testen Sie die Event-Funktion



Rufen Sie Ihre Lambda-Ereignisfunktion manuell mit dem `invoke` AWS CLI Befehl und einem Amazon SQS SQS-Beispielereignis auf.

So rufen Sie die Lambda-Ereignisfunktion mit einem Beispielereignis auf

1. Speichern Sie die folgende JSON als Datei mit dem Namen `input.json`. Dieses JSON simuliert ein Ereignis, das Amazon SQS möglicherweise an Ihre Lambda-Ereignisfunktion sendet, die die eigentliche Nachricht aus der Warteschlange `"body"` enthält. In diesem Beispiel lautet die Nachricht `"test"`.

   Example Amazon-SQS-Ereignis

   Dies ist ein Testereignis. Nachricht und Kontonummer müssen nicht geändert werden.

   ```
   {
       "Records": [
           {
               "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
               "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
               "body": "test",
               "attributes": {
                   "ApproximateReceiveCount": "1",
                   "SentTimestamp": "1545082649183",
                   "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                   "ApproximateFirstReceiveTimestamp": "1545082649185"
               },
               "messageAttributes": {},
               "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
               "eventSource": "aws:sqs",
   ```

```
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

2. Führen Sie den folgenden Invoke-Befehl aus. AWS CLI Dieser Befehl gibt CloudWatch Logs in der Antwort zurück. Weitere Informationen über das Abrufen von Protokollen finden Sie unter Zugriffs-Logs mit dem AWS CLI.

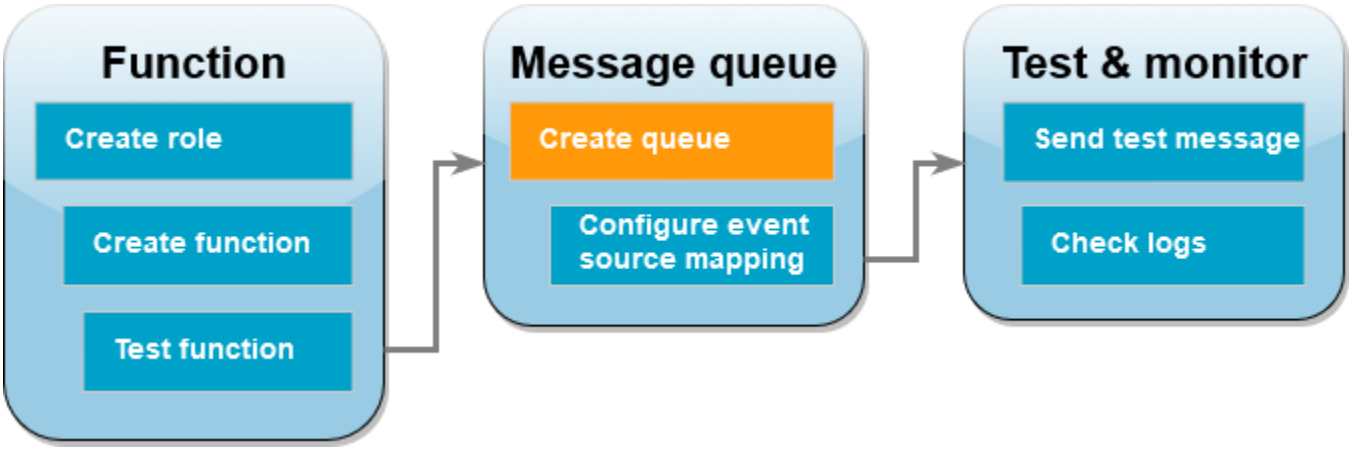
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

   Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

3. Suchen Sie in der Antwort nach dem Protokoll `INFO`. Hier protokolliert die Lambda-Ereignisfunktion den Nachrichtentext. Sie sollten Protokolle sehen, die wie folgt aussehen:

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

Erstellen einer Amazon-SQS-Warteschlange



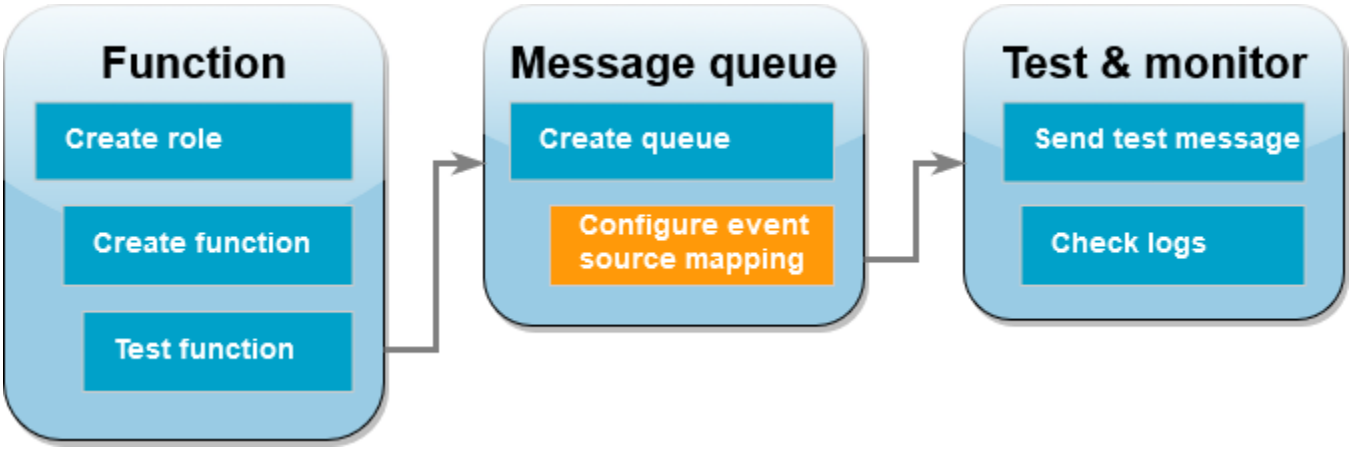
Erstellen Sie eine Amazon SQS SQS-Warteschlange, die die Lambda-Ereignisfunktion als Ereignisquelle verwenden kann. Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden. AWS-Region

So erstellen Sie eine Warteschlange

1. Öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie Create queue (Warteschlange erstellen) aus.
3. Geben Sie einen Namen für die Warteschlange ein. Übernehmen Sie bei allen anderen Optionen die Standardeinstellungen.
4. Wählen Sie Create queue (Warteschlange erstellen) aus.

Notieren Sie sich nach dem Erstellen der Warteschlange ihren ARN. Sie benötigen dies im nächsten Schritt, wenn Sie die Warteschlange mit Ihrer Lambda-Ereignisfunktion verknüpfen.

Konfigurieren der Ereignisquelle



Connect die Amazon SQS SQS-Warteschlange mit Ihrer Lambda-Ereignisfunktion, indem Sie eine Ereignisquellenzuordnung erstellen. Die Zuordnung der Ereignisquelle liest die Amazon SQS SQS-Warteschlange und ruft Ihre Lambda-Ereignisfunktion auf, wenn eine neue Nachricht hinzugefügt wird.

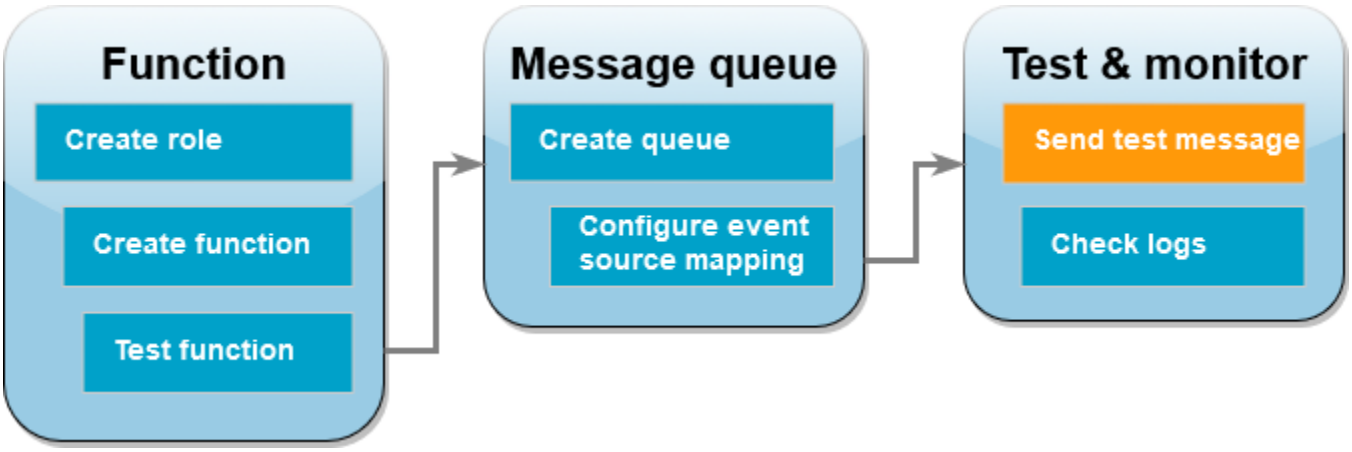
Verwenden Sie den Befehl create-event-source-mapping, um eine Zuordnung zwischen Ihrer Amazon SQS SQS-Warteschlange und Ihrer Lambda Event Function zu erstellen AWS CLI . Beispiel:

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

Verwenden Sie den Befehl list-event-source-mappings, um eine Liste Ihrer Zuordnungen von Ereignisquellen abzurufen. Beispiel:

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

Senden einer Testnachricht

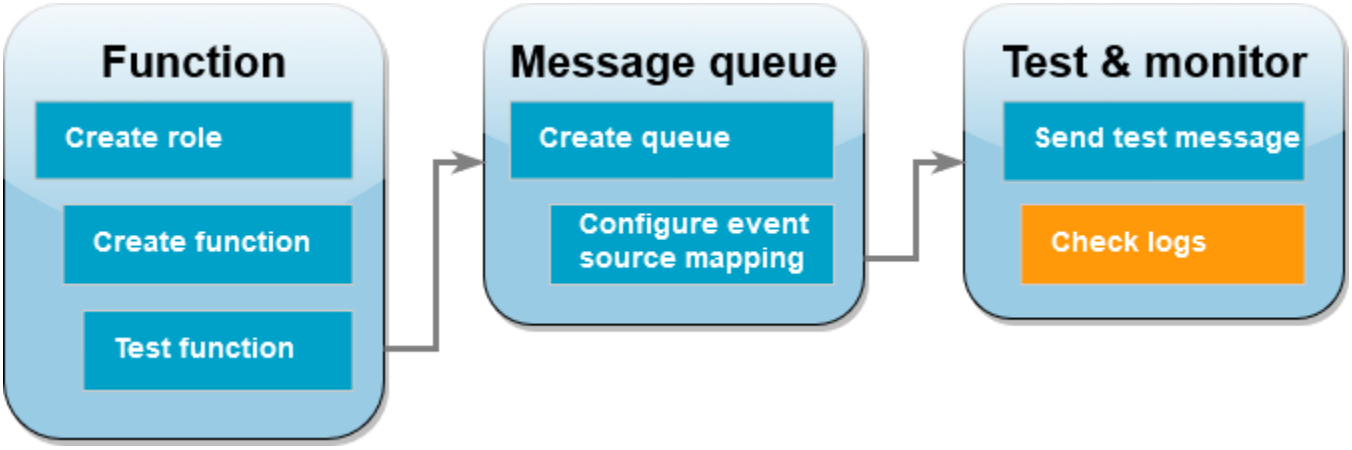


Um eine Amazon SQS SQS-Nachricht an die Lambda Event Function zu senden

1. Öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie die Warteschlange aus, die Sie zuvor erstellt haben.
3. Wählen Sie Nachrichten senden und empfangen.
4. Geben Sie unter Nachrichtentext eine Testnachricht ein (z. B. „Dies ist eine Testnachricht.“).
5. Klicken Sie auf Send Message (Nachricht senden).

Lambda fragt die Warteschlange nach Aktualisierungen ab. Wenn eine neue Nachricht vorliegt, ruft Lambda Ihre Funktion mit diesen neuen Ereignisdaten aus der Warteschlange auf. Wenn der Funktions-Handler ohne Ausnahmen zurückgegeben wird, betrachtet Lambda die Nachricht als erfolgreich verarbeitet und beginnt mit dem Lesen neuer Nachrichten in der Warteschlange. Nach erfolgreicher Verarbeitung einer Nachricht löscht Lambda diese automatisch aus der Warteschlange. Wenn der Handler eine Ausnahme auslöst, betrachtet Lambda den Nachrichten-Batch als nicht erfolgreich verarbeitet und Lambda ruft die Funktion mit demselben Nachrichten-Batch auf.

Überprüfen Sie die Protokolle CloudWatch



Um zu bestätigen, dass die Event Function die Nachricht verarbeitet hat

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion ProcessSQSRecord aus.
3. Wählen Sie Überwachen aus.
4. Wählen Sie CloudWatch Protokolle anzeigen aus.
5. Wählen Sie in der CloudWatch Konsole den Protokollstream für die Funktion aus.
6. Suchen Sie das Protokoll `INFO`. Hier protokolliert die Lambda-Ereignisfunktion den Nachrichtentext. Sie sollten die Nachricht sehen, die Sie über die Amazon-SQS-Warteschlange gesendet haben. Beispiel:

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.

4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Amazon-SQS-Warteschlange

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die Amazon SQS SQS-Konsole unter https://console.aws.amazon.com/sqs/.
2. Wählen Sie die Warteschlange aus, die Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

Tutorial: Verwenden einer kontoübergreifenden Amazon SQS SQS-Warteschlange als Ereignisquelle

In diesem Tutorial erstellen Sie eine Lambda-Ereignisfunktion, die Nachrichten aus einer Amazon Simple Queue Service (Amazon SQS) -Warteschlange in einem anderen Konto verarbeitet. AWS Dieses Tutorial umfasst zwei AWS Konten: Konto A bezieht sich auf das Konto, das Ihre Lambda-Ereignisfunktion enthält, und Konto B bezieht sich auf das Konto, das die Amazon SQS SQS-Warteschlange enthält.

Voraussetzungen

Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle (Konto A)

Erstellen Sie in Konto A eine Ausführungsrolle, die Ihrer Funktion die Erlaubnis erteilt, auf die erforderlichen AWS Ressourcen zuzugreifen.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Rollen in der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Vertrauenswürdige Entität – AWS Lambda.
   - Berechtigungen – AWSLambdaSQSQueueExecutionRole
   - Role name (Name der Rolle – **`cross-account-lambda-sqs-role`**

Die AWSLambdaSQSQueueExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion zum Lesen von Elementen aus Amazon SQS und zum Schreiben von Protokollen in Amazon CloudWatch Logs benötigt.

Erstellen Sie die Event-Funktion (Konto A)

Erstellen Sie in Konto A eine Lambda-Ereignisfunktion, die Ihre Amazon SQS SQS-Nachrichten verarbeitet. Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden. AWS-Region

Das folgende Node.js Codebeispiel schreibt jede Nachricht in ein Protokoll in CloudWatch Logs.

Example index.mjs

```
export const handler = async function(event, context) {
```

```
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

Um die Event-Funktion zu erstellen

> **Note**
>
> Wenn Sie diese Schritte befolgen, wird eine Node.js Funktion erstellt. Für andere Sprachen sind die Schritte ähnlich, aber einige Details unterscheiden sich.

1. Speichern Sie das Codebeispiel als Datei mit dem Namen `index.mjs`.
2. Erstellen Sie ein Bereitstellungspaket.

   ```
   zip function.zip index.mjs
   ```

3. Erstellen Sie die Funktion mit dem Befehl `create-function` AWS Command Line Interface (AWS CLI). Ersetzen Sie `arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role` durch den ARN der Ausführungsrolle, die Sie zuvor erstellt haben.

   ```
   aws lambda create-function --function-name CrossAccountSQSExample \
   --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
   --role arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role
   ```

Testen Sie die Event-Funktion (Konto A)

Testen Sie in Konto A Ihre Lambda-Ereignisfunktion manuell mithilfe des `invoke` AWS CLI Befehls und eines Amazon SQS SQS-Beispielereignisses.

Wenn der Handler normal und ohne Ausnahmen zurückkehrt, betrachtet Lambda die Nachricht als erfolgreich verarbeitet und beginnt mit dem Lesen neuer Nachrichten in der Warteschlange. Nach erfolgreicher Verarbeitung einer Nachricht löscht Lambda diese automatisch aus der Warteschlange. Wenn der Handler eine Ausnahme auslöst, betrachtet Lambda den Nachrichten-Batch als nicht erfolgreich verarbeitet und Lambda ruft die Funktion mit demselben Nachrichten-Batch auf.

1. Speichern Sie die folgende JSON als Datei mit dem Namen `input.txt`.

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

Das vorhergehende JSON simuliert ein Ereignis, das Amazon SQS möglicherweise an Ihre Lambda-Ereignisfunktion sendet, die die eigentliche Nachricht aus der Warteschlange `"body"` enthält.

2. Führen Sie den Befehl `invoke` AWS CLI aus.

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden AWS CLI . Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

3. Überprüfen Sie die Ausgabe in der Datei `outputfile.txt`.

Erstellen einer Amazon SQS-Warteschlange (Konto B)

Erstellen Sie in Konto B eine Amazon SQS SQS-Warteschlange, die die Lambda-Ereignisfunktion in Konto A als Ereignisquelle verwenden kann. Die Lambda-Ereignisfunktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden. AWS-Region

So erstellen Sie eine Warteschlange

1. Öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie Create queue (Warteschlange erstellen) aus.
3. Erstellen Sie eine Warteschlange mit den folgenden Eigenschaften.
   - Typ – Standard
   - Name (Name – LambdaCrossAccountQueue
   - Konfiguration – Behalten Sie die Standardeinstellungen bei.
   - Zugriffsrichtlinie – Wählen Sie Advanced (Erweitert). Fügen Sie die folgende JSON-Richtlinie ein. Ersetzen Sie die folgenden Werte:
     - 111122223333: AWS-Konto ID für Konto A
     - 444455556666: AWS-Konto ID für Konto B

     JSON

```
{
    "Version":"2012-10-17",
    "Id": "Queue1_Policy_UUID",
    "Statement": [
        {
            "Sid": "Queue1_AllActions",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:role/cross-account-lambda-
sqs-role"
                ]
            },
            "Action": "sqs:*",
            "Resource": "arn:aws:sqs:us-
east-1:444455556666:LambdaCrossAccountQueue"
        }
    ]
```

```
}
```

Diese Richtlinie gewährt der Lambda-Ausführungsrolle in Konto A Berechtigungen zur Nutzung von Nachrichten aus dieser Amazon-SQS-Warteschlange.

4. Zeichnen Sie nach dem Erstellen der Warteschlange ihren Amazon-Ressourcennamen (ARN) auf. Sie benötigen dies im nächsten Schritt, wenn Sie die Warteschlange mit Ihrer Lambda-Ereignisfunktion verknüpfen.

Konfigurieren Sie die Ereignisquelle (Konto A)

Erstellen Sie in Konto A eine Zuordnung der Ereignisquelle zwischen der Amazon SQS SQS-Warteschlange in Konto B und Ihrer Lambda-Ereignisfunktion, indem Sie den folgenden `create-event-source-mapping` AWS CLI Befehl ausführen. Ersetzen Sie `arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue` durch den ARN der Amazon-SQS-Warteschlange, die Sie im vorherigen Schritt erstellt haben.

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-size 10 \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Führen Sie den folgenden Befehl aus, um eine Liste Ihrer Ereignisquellen-Zuweisung abzurufen.

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Testen der Einrichtung

Sie können die Einrichtung nun wie folgt testen:

1. In Konto B öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie LambdaCrossAccountQueue, was Sie zuvor erstellt haben.
3. Wählen Sie Nachrichten senden und empfangen.
4. Geben Sie unter Nachrichtentext eine Testnachricht ein.
5. Klicken Sie auf Send Message (Nachricht senden).

Ihre Lambda-Event-Funktion in Konto A sollte die Nachricht erhalten. Lambda fragt die Warteschlange weiterhin nach Updates ab. Wenn eine neue Nachricht vorliegt, ruft Lambda Ihre

Funktion mit diesen neuen Ereignisdaten aus der Warteschlange auf. Ihre Funktion wird in Amazon ausgeführt und erstellt Protokolle CloudWatch. Sie können die Protokolle in der CloudWatch -Konsole ansehen.

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

Bereinigen Sie in Konto A Ihre Ausführungsrolle und die Lambda-Ereignisfunktion.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

In :Konto BBereinigen Sie die Amazon SQS SQS-Warteschlange.

So löschen Sie die Amazon-SQS-Warteschlange

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die Amazon SQS SQS-Konsole unter https://console.aws.amazon.com/sqs/.
2. Wählen Sie die Warteschlange aus, die Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

Orchestrierung von Lambda-Eventfunktionen mit Step Functions

AWS Step Functions bietet visuelle Workflow-Orchestrierung für die Koordination von Lambda Event Functions mit anderen Diensten. AWS Mit nativen Integrationen für mehr als 220 AWS Services und einer vollständig verwalteten, wartungsfreien Infrastruktur ist Step Functions ideal, wenn Sie visuelles Workflow-Design und vollständig verwaltete Serviceintegrationen benötigen.

Für die Orchestrierung mithilfe von Standardprogrammiersprachen innerhalb von Lambda, bei denen die Workflow-Logik neben der Geschäftslogik existiert, sollten Sie langlebige Lambda-Funktionen in Betracht ziehen. Hilfe bei der Auswahl zwischen diesen Optionen finden Sie unter Dauerhafte Funktionen oder Step Functions.

Beispielsweise kann die Bearbeitung eines Auftrags die Bestätigung der Auftragsdetails, die Überprüfung der Lagerbestände, die Bearbeitung der Zahlung und die Erstellung einer Rechnung erfordern. Schreiben Sie separate Lambda-Ereignisfunktionen für jede Aufgabe und verwenden Sie Step Functions, um den Workflow zu verwalten. Step Functions koordiniert den Datenfluss zwischen Ihren Funktionen und behandelt Fehler bei jedem Schritt. Diese Trennung macht es einfacher, Ihre Workflows zu visualisieren, zu ändern und zu verwalten, wenn sie immer komplexer werden.

Wann sollte Step Functions mit Lambda verwendet werden?

Die folgenden Szenarien sind gute Beispiele dafür, wann Step Functions besonders gut für die Orchestrierung von Lambda-based Anwendungen geeignet ist.

- Sequenzielle Verarbeitung
- Komplexe Fehlerbehandlung
- Bedingte Workflows und Genehmigungen durch Menschen
- Parallele Verarbeitung

Sequenzielle Verarbeitung

Bei sequenzieller Verarbeitung muss eine Aufgabe abgeschlossen sein, bevor die nächste Aufgabe beginnen kann. In einem Auftragsverarbeitungssystem kann die Zahlungsabwicklung beispielsweise erst beginnen, wenn die Auftragsvalidierung abgeschlossen ist, und die Rechnungsstellung muss auf die Zahlungsbestätigung warten. Schreiben Sie separate Lambda-Ereignisfunktionen für jede Aufgabe und verwenden Sie Step Functions, um die Sequenz zu verwalten und den Datenfluss zwischen Funktionen zu handhaben.

Anti-pattern Beispiel

Eine einzige Lambda-Event-Funktion verwaltet den gesamten Auftragsabwicklungsablauf durch:

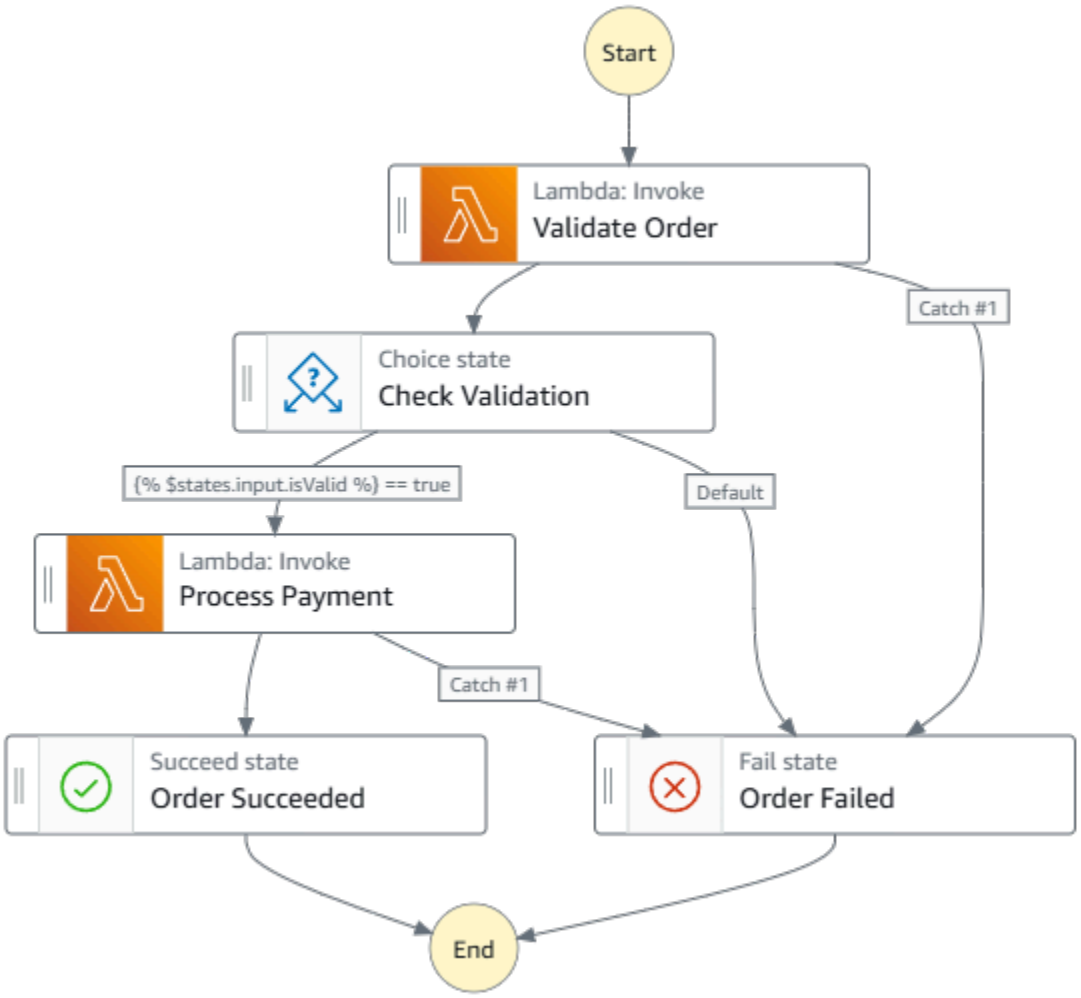
- Nacheinander andere Lambda-Ereignisfunktionen aufrufen
- Analysieren und Validieren der Antworten jeder Funktion
- Implementierung von Fehlerbehandlungs- und Wiederherstellungslogik
- Verwaltung des Datenflusses zwischen den Funktionen

Empfohlener Ansatz

Verwenden Sie zwei Lambda-Ereignisfunktionen: eine zur Bestätigung der Bestellung und eine zur Zahlungsabwicklung. Step Functions koordiniert diese Funktionen durch Folgendes:

- Ausführung von Aufgaben in der richtigen Reihenfolge
- Weitergabe von Daten zwischen Funktionen
- Implementierung der Fehlerbehandlung bei jedem Schritt
- Verwendung von Choice-Zuständen, um sicherzustellen, dass nur gültige Aufträge zur Zahlung weitergeleitet werden

Example Workflow-Diagramm



> **Note**
>
> Code-first Alternative: Informationen zur sequentiellen Verarbeitung mit codebasiertem Checkpointing und Wiederholungsversuch finden Sie unter Lambda Durable Functions Steps.

Komplexe Fehlerbehandlung

Während Lambda Wiederholungsfunktionen für asynchrone Aufrufe und Zuordnungen von Ereignisquellen bietet, bietet Step Functions eine ausgefeiltere Fehlerbehandlung für komplexe Workflows. Sie können automatische Wiederholungsversuche mit exponentiellem Backoff konfigurieren und unterschiedliche Wiederholungsrichtlinien für verschiedene Fehlertypen festlegen. Wenn alle Wiederholungsversuche ausgeschöpft sind, verwenden Sie `Catch`, um Fehler in einen

Fallback-Zustand umzuleiten. Dies ist besonders nützlich, wenn Sie eine Fehlerbehandlung auf Workflow-Ebene benötigen, die mehrere Funktionen und Services koordiniert.

Weitere Informationen zur Behandlung von Fehlern in der Lambda-Ereignisfunktion in einer Zustandsmaschine finden Sie unter Behandlung von Fehlern in The AWS Step Functions Workshop.

Anti-pattern Beispiel

Eine einzige Lambda-Ereignisfunktion behandelt alle folgenden Aufgaben:

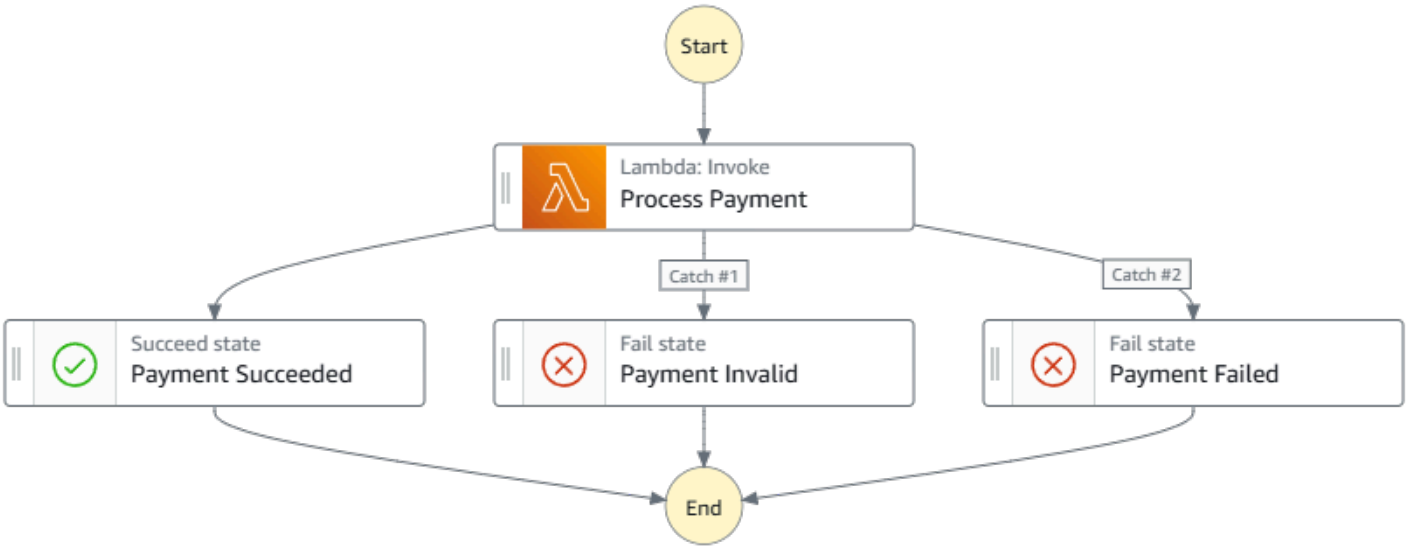
- Aufrufversuche eines Zahlungsabwicklungsdienstes
- Wenn der Zahlungsdienst nicht verfügbar ist, wartet die Funktion und versucht es später erneut.
- Implementierung eines benutzerdefinierten exponentiellen Backoffs für die Wartezeit
- Nachdem alle Versuche fehlgeschlagen sind, wird der Fehler abgefangen und ein anderer Ablauf gewählt.

Empfohlener Ansatz

Verwenden Sie eine einzige Lambda-Event-Funktion, die sich ausschließlich auf die Zahlungsabwicklung konzentriert. Step Functions verwaltet die Fehlerbehandlung durch Folgendes:

- Automatische Wiederholung fehlgeschlagener Aufgaben mit konfigurierbaren Backoff-Zeiten
- Anwendung unterschiedlicher Wiederholungsrichtlinien je nach Fehlertyp
- Weiterleitung verschiedener Fehlertypen an geeignete Fallback-Zustände
- Beibehaltung des Status und des Verlaufs der Fehlerbehandlung

Example Workflow-Diagramm



> **Note**
>
> Code-first Alternative: Dauerhafte Funktionen ermöglichen die Behandlung von Try-Catch-Fehlern mit konfigurierbaren Wiederholungsstrategien. Siehe Fehlerbehandlung bei dauerhaften Funktionen.

Bedingte Workflows und Genehmigungen durch Menschen

Verwenden Sie den Status Step Functions Choice, um Workflows auf der Grundlage der Funktionsausgabe weiterzuleiten, und verwenden Sie das ForTaskToken Wartesuffix, um Workflows für menschliche Entscheidungen anzuhalten. Um beispielsweise eine Anfrage zur Erhöhung des Kreditlimits zu bearbeiten, verwenden Sie eine Lambda-Ereignisfunktion, um Risikofaktoren zu bewerten. Verwenden Sie dann Step Functions, um Anfragen mit hohem Risiko an die manuelle Genehmigung und Anfragen mit geringem Risiko an die automatische Genehmigung weiterzuleiten.

Informationen zur Bereitstellung eines Beispiel-Workflows, der ein Integrationsmuster für Callback-Task-Tokens verwendet, finden Sie unter Callback with Task Token in The Workshop. AWS Step Functions

Anti-pattern Beispiel

Eine einzelne Lambda-Ereignisfunktion verwaltet einen komplexen Genehmigungsworkflow durch:

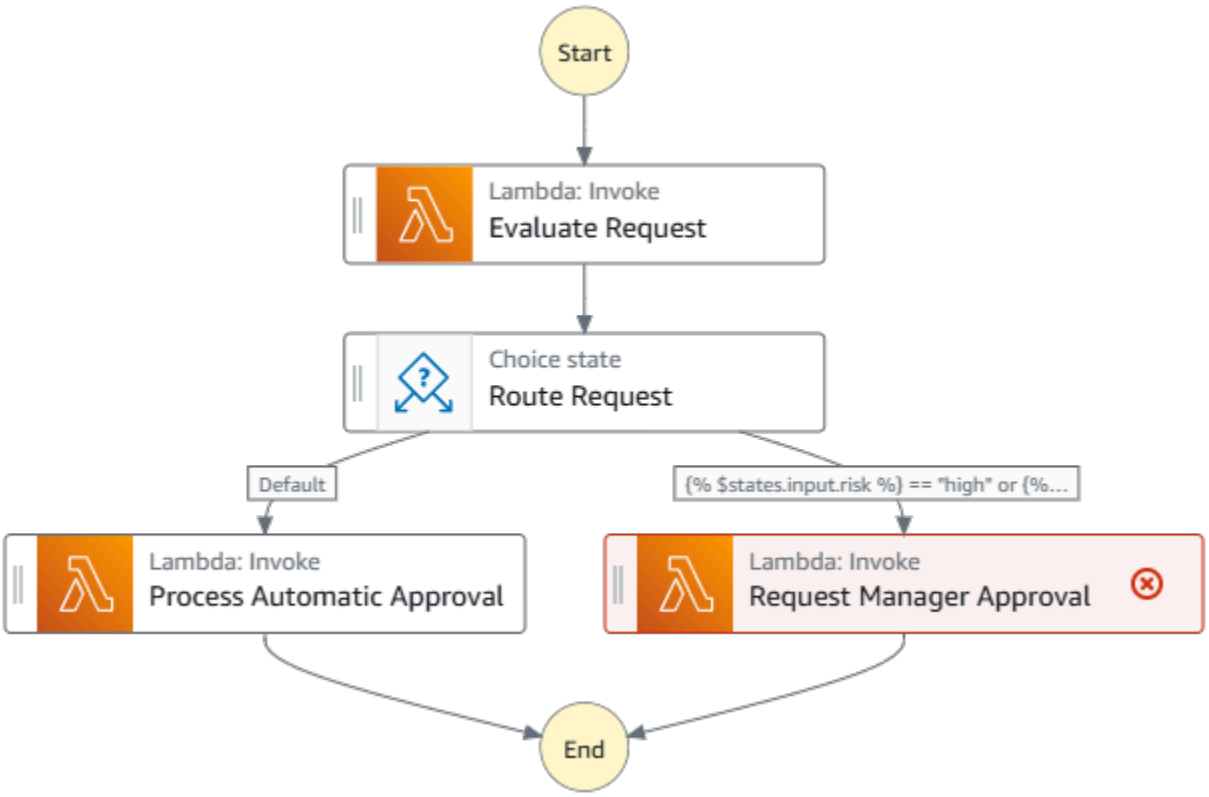
- Implementierung einer verschachtelten bedingten Logik zur Bewertung von Kreditanfragen
- Aufrufen verschiedener Genehmigungsfunktionen auf der Grundlage der Anzahl der Anfragen
- Verwaltung mehrerer Genehmigungspfade und Entscheidungspunkte
- Nachverfolgung des Status ausstehender Genehmigungen
- Implementierung einer Timeout- und Benachrichtigungslogik für Genehmigungen

Empfohlener Ansatz

Verwenden Sie drei Lambda-Ereignisfunktionen: eine, um das Risiko jeder Anfrage zu bewerten, eine, um Anfragen mit geringem Risiko zu genehmigen, und eine, um Anfragen mit hohem Risiko zur Überprüfung an einen Manager weiterzuleiten. Step Functions steuert den Workflow durch:

- Verwendung von Choice-Zuständen zur Weiterleitung von Anfragen basierend auf Betrag und Risikostufe
- Pausieren der Ausführung bis zur Genehmigung durch einen Menschen
- Verwaltung von Timeouts für ausstehende Genehmigungen
- Bereitstellung von Einblicken in den aktuellen Status jeder Anfrage

Example Workflow-Diagramm



> **Note**
>
> Code-first Alternative: Dauerhafte Funktionen unterstützen Callbacks für Workflows, bei denen ein Mensch im Kreislauf arbeitet. Siehe Callbacks in dauerhaften Funktionen.

Parallele Verarbeitung

Step Functions bietet drei Möglichkeiten, die Parallelverarbeitung zu handhaben:

- Im parallelen Status werden mehrere Zweige Ihres Workflows gleichzeitig ausgeführt. Verwenden Sie diese Option, wenn Sie verschiedene Funktionen parallel ausführen müssen, z. B. das Generieren von Miniaturansichten beim Extrahieren von Bildmetadaten.
- Der Inline-Zuordnungszustand verarbeitet Datenarrays mit bis zu 40 gleichzeitigen Iterationen. Verwenden Sie dies für kleine bis mittlere Datensätze, bei denen Sie für jedes Element dieselbe Operation ausführen müssen.

- Der Zustand „Verteilte Karte“ verarbeitet umfangreiche Parallelverarbeitung mit bis zu 10 000 gleichzeitigen Ausführungen und unterstützt sowohl JSON-Arrays als auch Datenquellen von Amazon Simple Storage Service (Amazon S3). Verwenden Sie diese Option, wenn Sie große Datensätze verarbeiten oder wenn Sie eine höhere Gleichzeitigkeit benötigen.

Anti-pattern Beispiel

Eine einzelne Lambda-Ereignisfunktion versucht, die Parallelverarbeitung wie folgt zu verwalten:

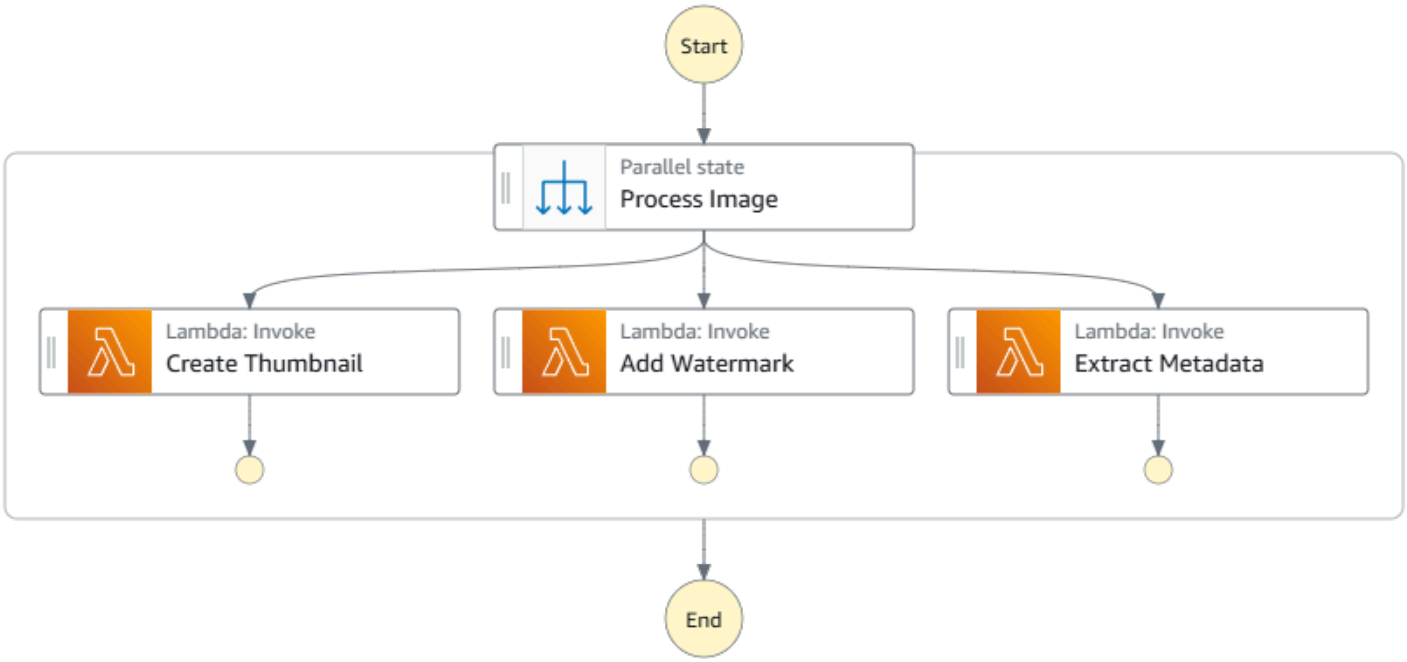
- Gleichzeitiges Aufrufen mehrerer Bildverarbeitungsfunktionen
- Implementierung einer benutzerdefinierten Logik für die parallel Ausführung
- Verwaltung von Timeouts und Fehlerbehandlung für jede parallele Aufgabe
- Sammeln und Aggregieren von Ergebnissen aus allen Funktionen

Empfohlener Ansatz

Verwenden Sie drei Lambda-Ereignisfunktionen: eine zum Erstellen eines Miniaturbilds, eine zum Hinzufügen eines Wasserzeichens und eine zum Extrahieren der Metadaten. Step Functions verwaltet diese Funktionen wie folgt:

- Gleichzeitiges Ausführen aller Funktionen im parallelen Status
- Sammeln der Ergebnisse jeder Funktion in einem geordneten Array
- Verwaltung von Timeouts und Fehlerbehandlung bei allen parallelen Ausführungen
- Fortsetzung erst nach Abschluss aller parallelen Zweige

Example Workflow-Diagramm



> **Note**
>
> Code-first Alternative: Dauerhafte Funktionen bieten `parallel()` und funktionieren. `map()` Siehe Parallele Ausführung.

Wann sollte Step Functions mit Lambda nicht verwendet werden?

Nicht alle Lambda-based Anwendungen profitieren von der Verwendung von Step Functions. Berücksichtigen Sie diese Szenarien bei der Auswahl Ihrer Anwendungsarchitektur.

- Einfache Anwendungen
- Komplexe Datenverarbeitung
- CPU-intensive Arbeitslasten

Einfache Anwendungen

> **Note**
>
> Für Workflows, die kein visuelles Design oder umfangreiche Serviceintegrationen erfordern, könnten langlebige Lambda-Funktionen eine einfachere Alternative sein, bei der die Workflow-Logik im Code innerhalb von Lambda beibehalten wird.

Bei Anwendungen, die keine komplexe Orchestrierung erfordern, kann die Verwendung von Step Functions zu unnötiger Komplexität führen. Wenn Sie beispielsweise einfach Nachrichten aus einer Amazon SQS SQS-Warteschlange verarbeiten oder auf EventBridge Amazon-Ereignisse reagieren, können Sie diese Dienste so konfigurieren, dass sie Ihre Lambda-Ereignisfunktionen direkt aufrufen. Wenn Ihre Anwendung nur aus einer oder zwei Lambda-Ereignisfunktionen mit einfacher Fehlerbehandlung besteht, sind direkte Lambda-Aufrufe oder ereignisgesteuerte Architekturen möglicherweise einfacher bereitzustellen und zu verwalten.

Komplexe Datenverarbeitung

Sie können den Status Step Functions Distributed Map verwenden, um große Amazon S3 S3-Datensätze gleichzeitig mit Lambda Event Functions zu verarbeiten. Dies ist für viele umfangreiche parallele Workloads effektiv, einschließlich der Verarbeitung halbstrukturierter Daten wie JSON- oder CSV-Dateien. Für komplexere Datentransformationen oder erweiterte Analytik sollten Sie jedoch die folgenden Alternativen in Betracht ziehen:

- Pipelines zur Datentransformation: Wird AWS Glue für ETL-Jobs verwendet, die strukturierte oder halbstrukturierte Daten aus mehreren Quellen verarbeiten. AWS Glue ist besonders nützlich, wenn Sie integrierte Datenkatalog- und Schemaverwaltungsfunktionen benötigen.
- Datenanalytik: Verwenden Sie Amazon EMR für Datenanalytik im Petabyte-Bereich, insbesondere wenn Sie Tools des Apache Hadoop-Ökosystems benötigen oder für Machine-Learning-Workloads, die die Speichergrenzen von Lambda überschreiten.

CPU-intensive Arbeitslasten

Step Functions können zwar CPU-intensive Aufgaben orchestrieren, Lambda Event Functions sind jedoch aufgrund ihrer begrenzten CPU-Ressourcen möglicherweise nicht für diese Workloads geeignet. Für rechenintensive Operationen innerhalb Ihrer Workflows sollten Sie die folgenden Alternativen in Betracht ziehen:

- Container-Orchestrierung: Verwenden Sie Step Functions zur Verwaltung von Aufgaben von Amazon Elastic Container Service (Amazon ECS) für konsistentere und skalierbarere Rechenressourcen.
- Stapelverarbeitung: Integrieren Sie AWS Batch Step Functions für die Verwaltung rechenintensiver Batch-Jobs, die eine kontinuierliche CPU-Auslastung erfordern.

Rufen Sie eine Lambda-Ereignisfunktion mit Amazon S3-Batchereignissen auf

Sie können Amazon S3 S3-Batchoperationen verwenden, um eine Lambda-Ereignisfunktion für eine große Menge von Amazon S3 S3-Objekten aufzurufen. Amazon S3 verfolgt den Fortschritt von Batchvorgängen, sendet Benachrichtigungen und speichert einen Abschlussbericht, der den Status jeder Aktion anzeigt.

Zum Ausführen eines Batchvorgangs erstellen Sie einen Amazon-S3-Batchvorgangsauftrag. Wenn Sie den Auftrag erstellen, stellen Sie ein Manifest (die Liste der Objekte) bereit und konfigurieren die Aktion für diese Objekte.

Wenn der Batch-Job gestartet wird, ruft Amazon S3 die Lambda-Ereignisfunktion synchron für jedes Objekt im Manifest auf. Der Ereignis-Parameter enthält die Namen des Buckets und des Objekts.

Das folgende Beispiel zeigt das Ereignis, das Amazon S3 an die Lambda-Ereignisfunktion für ein Objekt mit dem Namen customer Image1.jpg im Bucket amzn-s3-demo-bucket sendet.

Example Batch-Anforderungsereignis für Amazon S3

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::amzn-s3-demo-bucket"
        }
    ]
}
```

Ihre Lambda-Ereignisfunktion muss ein JSON-Objekt mit den Feldern zurückgeben, wie im folgenden Beispiel gezeigt. Sie können den Parameter `invocationId` und `taskId` aus dem Ereignisparameter kopieren. Sie können eine Zeichenfolge in `resultString` zurückgeben. Amazon S3 speichert die `resultString`-Werte im Abschlussbericht.

Example Batch-Anforderungantwort für Amazon S3

```
{
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

Aufrufen von Lambda-Ereignisfunktionen aus Amazon S3 S3-Batch-Vorgängen

Sie können die Lambda-Ereignisfunktion mit einem unqualifizierten oder qualifizierten Funktions-ARN aufrufen. Wenn Sie dieselbe Funktionsversion für den gesamten Stapelauftrag verwenden möchten, konfigurieren Sie beim Erstellen des Auftrags eine bestimmte Funktionsversion im Parameter `FunctionARN`. Wenn Sie einen Alias oder den Qualifizierer $LATEST konfigurieren, beginnt der Stapelauftrag sofort, die neue Version der Funktion aufzurufen, wenn der Alias oder $LATEST während der Auftragsausführung aktualisiert wird.

Beachten Sie, dass Sie eine vorhandene ereignisbasierte Amazon-S3-Funktion nicht für Batchvorgänge verwenden können. Dies liegt daran, dass der Amazon S3 S3-Batchvorgang einen anderen Ereignisparameter an die Lambda-Ereignisfunktion übergibt und eine Rückmeldung mit einer bestimmten JSON-Struktur erwartet.

Stellen Sie in der ressourcenbasierten Richtlinie, die Sie für den Amazon S3 S3-Batch-Job erstellen, sicher, dass Sie für den Job die Berechtigung zum Aufrufen Ihrer Lambda-Ereignisfunktion festlegen.

Legen Sie in der Ausführungsrolle für die Funktion eine Vertrauensrichtlinie fest, damit Amazon S3 die Rolle übernimmt, wenn die Funktion ausgeführt wird.

Wenn Ihre Funktion das AWS SDK zur Verwaltung von Amazon S3 S3-Ressourcen verwendet, müssen Sie der Ausführungsrolle Amazon S3 S3-Berechtigungen hinzufügen.

Wenn der Auftrag ausgeführt wird, startet Amazon S3 mehrere Funktionsinstances, um die Amazon-S3-Objekte bis zur Parallelitätsgrenze der Funktion parallel zu verarbeiten. Amazon S3 begrenzt den anfänglichen Hochlauf von Instances, um Überkosten für kleinere Aufträge zu vermeiden.

Wenn die Lambda-Ereignisfunktion einen `TemporaryFailure` Antwortcode zurückgibt, versucht Amazon S3 den Vorgang erneut.

Weitere Informationen zu Amazon-S3-Stapelvorgängen finden Sie unter Durchführen von Stapelvorgängen im Amazon-S3-Entwicklerhandbuch.

Ein Beispiel für die Verwendung einer Lambda-Ereignisfunktion in Amazon S3 S3-Batchvorgängen finden Sie unter Aufrufen einer Lambda-Funktion aus Amazon S3 S3-Batchoperationen im Amazon S3 S3-Entwicklerhandbuch.

Aufrufen von Lambda-Ereignisfunktionen mit Amazon SNS SNS-Benachrichtigungen

Sie können eine Lambda-Ereignisfunktion verwenden, um Amazon Simple Notification Service (Amazon SNS) -Benachrichtigungen zu verarbeiten. Amazon SNS unterstützt Lambda Event Functions als Ziel für Nachrichten, die an ein Thema gesendet werden. Sie können Ihre Funktion für Themen in demselben Konto oder in anderen AWS -Konten abonnieren. Eine ausführliche exemplarische Vorgehensweise finden Sie unter the section called “Tutorial”.

Lambda unterstützt SNS-Auslöser nur für standardmäßige SNS-Themen. FIFO-Themen werden nicht unterstützt.

Lambda verarbeitet SNS-Nachrichten asynchron, indem die Nachrichten in eine Warteschlange gestellt und Wiederholungsversuche verarbeitet werden. Wenn Amazon SNS Lambda nicht erreichen kann, oder die Nachricht abgelehnt wird, wiederholt Amazon SNS den Vorgang in zunehmenden Intervallen über mehrere Stunden. Weitere Details finden Sie unter Zuverlässigkeit in den häufig gestellten Fragen zu Amazon SNS.

> **Warning**
>
> Asynchrone Lambda-Aufrufe verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Das Hilfsprogramm Idempotenz von Powertools für AWS Lambda

Das Idempotenz-Hilfsprogramm von Powertools for AWS Lambda macht Ihre Lambda Event Functions idempotent. Es ist für Python TypeScript, Java und .NET verfügbar. Weitere Informationen finden Sie unter Idempotenz-Hilfsprogramm in der Powertools for AWS Lambda (Python) -Dokumentation, Idempotenz-Hilfsprogramm in der Powertools for AWS Lambda () -Dokumentation, Idempotenz-Hilfsprogramm in der Powertools for Lambda (JavaTypeScript) -Dokumentation und Idempotenz-Hilfsprogramm in der Powertools for AWS Lambda (.NET) -Dokumentation. AWS

Themen

- Hinzufügen eines Amazon SNS SNS-Themenauslösers für eine Lambda-Ereignisfunktion mithilfe der Konsole
- Manuelles Hinzufügen eines Amazon SNS SNS-Themenauslösers für eine Lambda-Ereignisfunktion
- Beispiel für eine SNS-Eventform
- Tutorial: Verwenden AWS Lambda mit Amazon Simple Notification Service

Hinzufügen eines Amazon SNS SNS-Themenauslösers für eine Lambda-Ereignisfunktion mithilfe der Konsole

Um ein SNS-Thema als Auslöser für eine Lambda-Ereignisfunktion hinzuzufügen, verwenden Sie am einfachsten die Lambda-Konsole. Wenn Sie den Trigger über die Konsole hinzufügen, richtet Lambda automatisch die erforderlichen Berechtigungen und Abonnements ein, um Ereignisse aus dem SNS-Thema zu empfangen.

So fügen Sie ein SNS-Thema als Auslöser für eine Lambda-Ereignisfunktion hinzu (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion, für die Sie den Auslöser hinzufügen möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie Add trigger.
5. Wählen Sie unter Auslöserkonfiguration im Dropdown-Menü die Option SNS aus.
6. Wählen Sie für das SNS-Thema das SNS-Thema, das Sie abonnieren möchten.

Manuelles Hinzufügen eines Amazon SNS SNS-Themenauslösers für eine Lambda-Ereignisfunktion

Um einen SNS-Trigger für eine Lambda-Ereignisfunktion manuell einzurichten, müssen Sie die folgenden Schritte ausführen:

- Definieren Sie eine ressourcenbasierte Richtlinie für Ihre Funktion, damit SNS sie aufrufen kann.
- Abonnieren Sie Ihre Lambda Event Function für das Amazon SNS SNS-Thema.

> **Note**
>
> Wenn sich Ihr SNS-Thema und Ihre Lambda-Ereignisfunktion in unterschiedlichen AWS Konten befinden, müssen Sie auch zusätzliche Berechtigungen gewähren, um kontoübergreifende Abonnements für das SNS-Thema zuzulassen. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigung für Amazon-SNS-Abonnement erteilen.

Sie können das AWS Command Line Interface (AWS CLI) verwenden, um diese beiden Schritte abzuschließen. Verwenden Sie zunächst den folgenden Befehl, um eine ressourcenbasierte Richtlinie für eine Lambda-Ereignisfunktion zu definieren, die SNS-Aufrufe zulässt. AWS CLI Achten Sie darauf, den Wert von `--function-name` durch den Namen Ihrer Lambda-Ereignisfunktion und den Wert von `--source-arn` durch den ARN Ihres SNS-Themas zu ersetzen.

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

Verwenden Sie den folgenden Befehl, um das SNS-Thema für Ihre Funktion zu abonnieren. AWS CLI Ersetzen Sie den Wert von `--topic-arn` durch Ihren SNS-Thema-ARN und den Wert von `--notification-endpoint` durch Ihren Lambda Event Function ARN.

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-
function
```

Beispiel für eine SNS-Eventform

Amazon SNS ruft Ihre Funktion asynchron mit einem Ereignis auf, das eine Nachricht und Metadaten enthält.

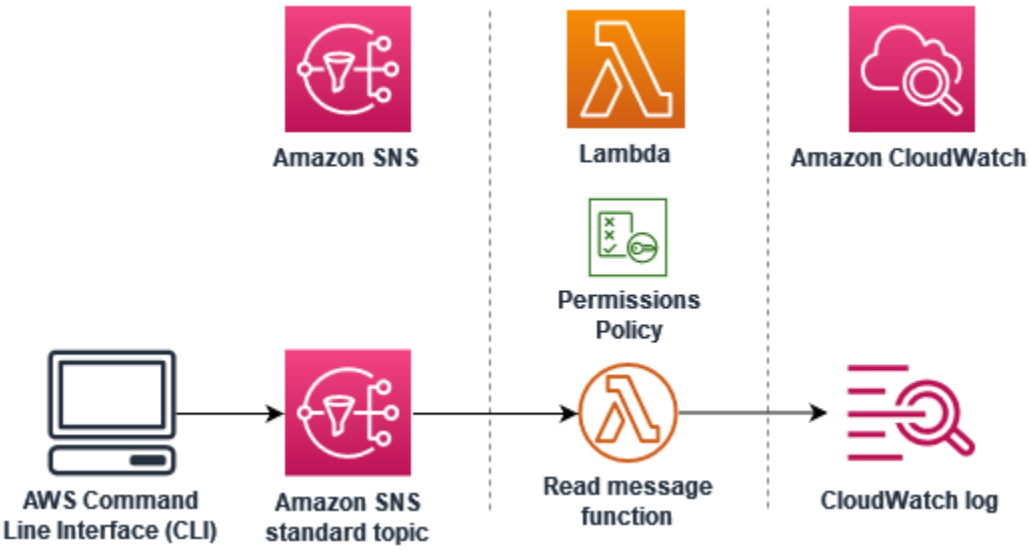
Example Amazon-SNS-Nachrichtenereignis

```
{
  "Records": [
```

```
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-
a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeUrl": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

Tutorial: Verwenden AWS Lambda mit Amazon Simple Notification Service

In diesem Tutorial verwenden Sie eine Lambda-Ereignisfunktion in einer, AWS-Konto um ein Amazon Simple Notification Service (Amazon SNS) -Thema in einem separaten zu abonnieren. AWS-Konto Wenn Sie Nachrichten zu Ihrem Amazon SNS SNS-Thema veröffentlichen, liest Ihre Lambda-Ereignisfunktion den Inhalt der Nachricht und gibt ihn in Amazon CloudWatch Logs aus. Um dieses Tutorial abzuschließen, verwenden Sie die AWS Command Line Interface ()AWS CLI.



Gehen Sie für dieses Tutorial wie folgt vor:

- Erstellen Sie in Konto A ein Amazon-SNS-Thema.
- Erstellen Sie in Konto B eine Lambda-Ereignisfunktion, die Nachrichten aus dem Thema liest.
- Erstellen Sie in Konto B ein Abonnement für das Thema.
- Veröffentlichen Sie Nachrichten zum Amazon SNS SNS-Thema in Konto A und stellen Sie sicher, dass die Lambda-Ereignisfunktion in Konto B sie in Logs ausgibt. CloudWatch

Wenn Sie diese Schritte ausführen, erfahren Sie, wie Sie ein Amazon SNS SNS-Thema so konfigurieren, dass es eine Lambda-Ereignisfunktion aufruft. Sie erfahren auch, wie Sie eine AWS Identity and Access Management (IAM-) Richtlinie erstellen, die einer Ressource in einer anderen die Erlaubnis erteilt AWS-Konto , Lambda aufzurufen.

In dem Tutorial werden zwei separate AWS-Konten verwendet. Die AWS CLI Befehle veranschaulichen dies anhand von zwei benannten Profilen`accountB`, die aufgerufen `accountA` und jeweils für die Verwendung mit einem anderen konfiguriert sind. AWS-Konto Informationen zur Konfiguration der AWS CLI Verwendung verschiedener Profile finden Sie unter Einstellungen für Konfiguration und Anmeldeinformationsdatei im AWS Command Line Interface Benutzerhandbuch für Version 2. Stellen Sie sicher, dass Sie AWS-Region für beide Profile dieselbe Standardeinstellung konfigurieren.

Wenn die AWS CLI Profile, die Sie für die beiden erstellen, unterschiedliche Namen AWS-Konten verwenden oder wenn Sie das Standardprofil und ein benanntes Profil verwenden, ändern Sie die AWS CLI Befehle in den folgenden Schritten nach Bedarf.

Voraussetzungen

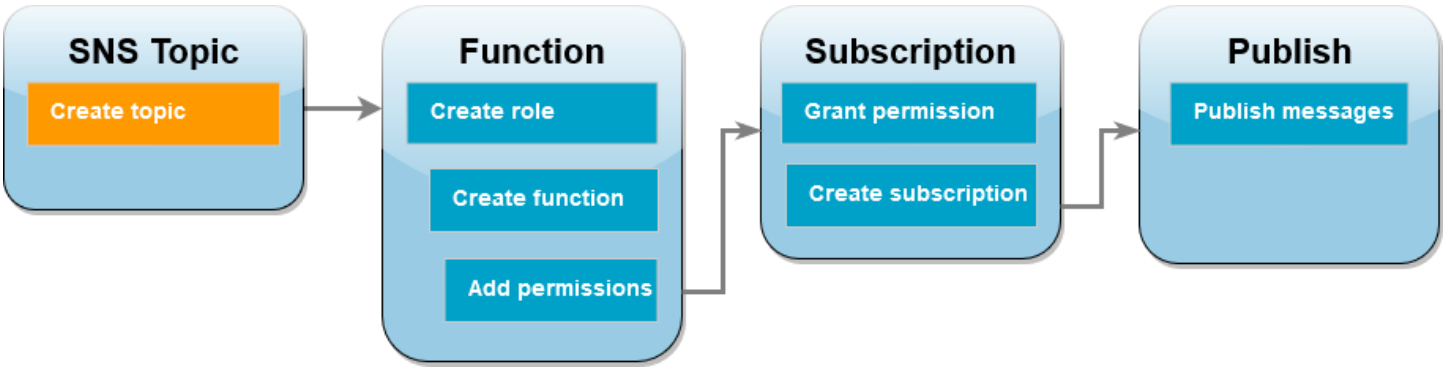
Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine Windows-integrated Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen eines Amazon-SNS-Themas (Konto A)



So erstellen Sie das -Thema

- Erstellen Sie in Konto A mit dem folgenden AWS CLI Befehl ein Amazon SNS SNS-Standardthema.

  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

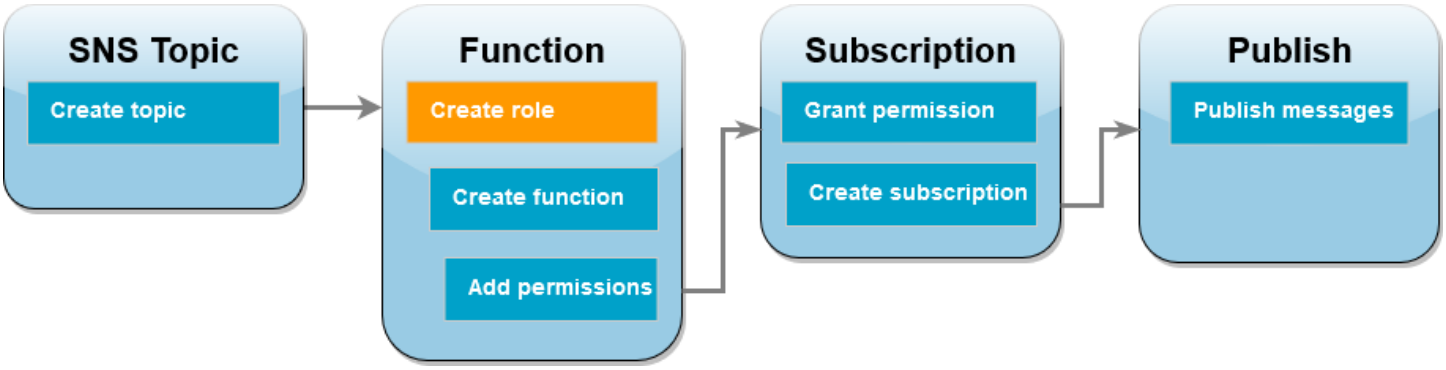
  Die Ausgabe sollte in etwa wie folgt aussehen:

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  ```

```
}
```

Notieren Sie sich den Amazon-Ressourcennamen (ARN) Ihres Themas. Sie benötigen es später im Tutorial, wenn Sie Ihrer Lambda-Ereignisfunktion Berechtigungen hinzufügen, um das Thema zu abonnieren.

Erstellen einer Funktionsausführungsrolle (Konto B)



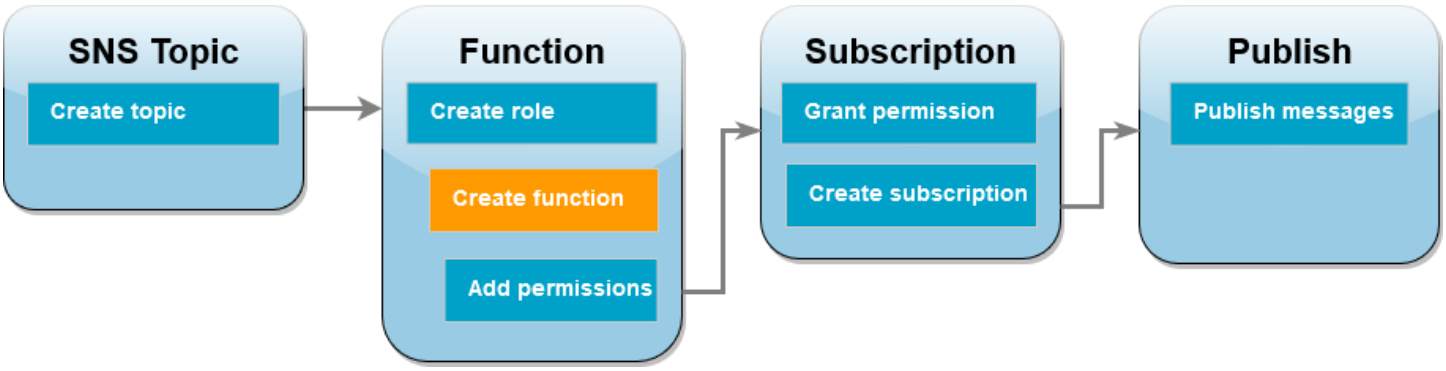
Eine Ausführungsrolle ist eine IAM-Rolle, die einer Lambda-Ereignisfunktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. Bevor Sie Ihre Funktion in Konto B erstellen, erstellen Sie eine Rolle, die der Funktion grundlegende Berechtigungen zum Schreiben von Protokollen in Logs erteilt. CloudWatch In einem späteren Schritt fügen wir die Berechtigungen zum Lesen aus Ihrem Amazon SNS SNS-Thema hinzu.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie in Konto B die Seite Rollen in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall die Option Lambda aus.
5. Wählen Sie Weiter aus.
6. Fügen Sie der Rolle wie folgt eine Richtlinie mit grundlegenden Berechtigungen hinzu:
   a. Geben Sie im Suchfeld Berechtigungsrichtlinien die Zeichenfolge **`AWSLambdaBasicExecutionRole`** ein.
   b. Wählen Sie Weiter aus.
7. Schließen Sie die Rollenerstellung ab:
   a. Geben Sie unter Rollendetails im Feld Rollenname den Namen **`lambda-sns-role`** ein.

b. Wählen Sie Rolle erstellen aus.

Erstellen Sie eine Lambda-Ereignisfunktion (Konto B)



Erstellen Sie eine Lambda-Ereignisfunktion, die Ihre Amazon SNS SNS-Nachrichten verarbeitet. Der Funktionscode protokolliert den Nachrichteninhalt jedes Datensatzes in Amazon CloudWatch Logs.

Dieses Tutorial verwendet die Node.js 24-Runtime, aber wir haben auch Beispielcode in anderen Runtime-Sprachen bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um Code für die gewünschte Laufzeit anzusehen. Der JavaScript Code, den Sie in diesem Schritt verwenden, befindet sich im ersten Beispiel, das auf der JavaScriptRegisterkarte angezeigt wird.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
```

```
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines SNS-Ereignisses mit Lambda unter Verwendung. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

Konsumieren eines SNS-Ereignisses mit Lambda unter Verwendung. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von PHP

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Rust

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
```

```
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Um die Event-Funktion zu erstellen

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir sns-tutorial
   cd sns-tutorial
   ```

2. Kopieren Sie den JavaScript Beispielcode in eine neue Datei mit dem Namen`index.js`.
3. Erstellen Sie ein Bereitstellungspaket mit dem folgenden `zip`-Befehl.

```
zip function.zip index.js
```

4. Führen Sie den folgenden AWS CLI Befehl aus, um Ihre Lambda-Ereignisfunktion in Konto B zu erstellen.

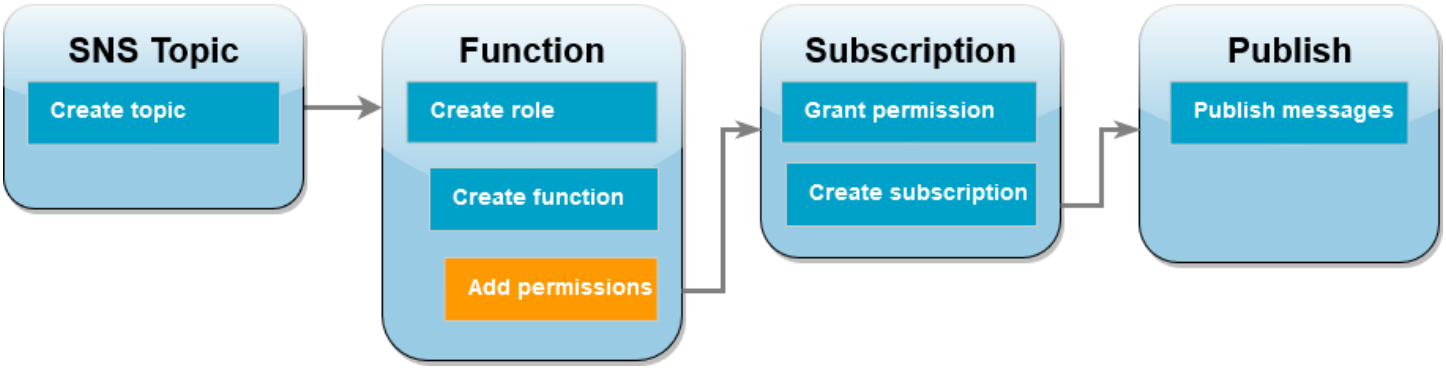
```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. Notieren Sie sich den Amazon-Ressourcennamen (ARN) Ihrer Funktion. Sie benötigen es später im Tutorial, wenn Sie Berechtigungen hinzufügen, damit Amazon SNS Ihre Funktion aufrufen kann.

Fügen Sie der Event-Funktion (Konto B) Berechtigungen hinzu



Damit Amazon SNS Ihre Funktion aufrufen kann, muss die dafür notwendige Berechtigung in einer Anweisung einer ressourcenbasierten Richtlinie erteilt werden. Sie fügen diese Anweisung mit dem AWS CLI `add-permission` Befehl hinzu.

Um Amazon SNS die Erlaubnis zu erteilen, Ihre Event-Funktion aufzurufen

- Führen Sie in Konto B den folgenden AWS CLI Befehl mit dem ARN für Ihr Amazon SNS SNS-Thema aus, das Sie zuvor aufgezeichnet haben.

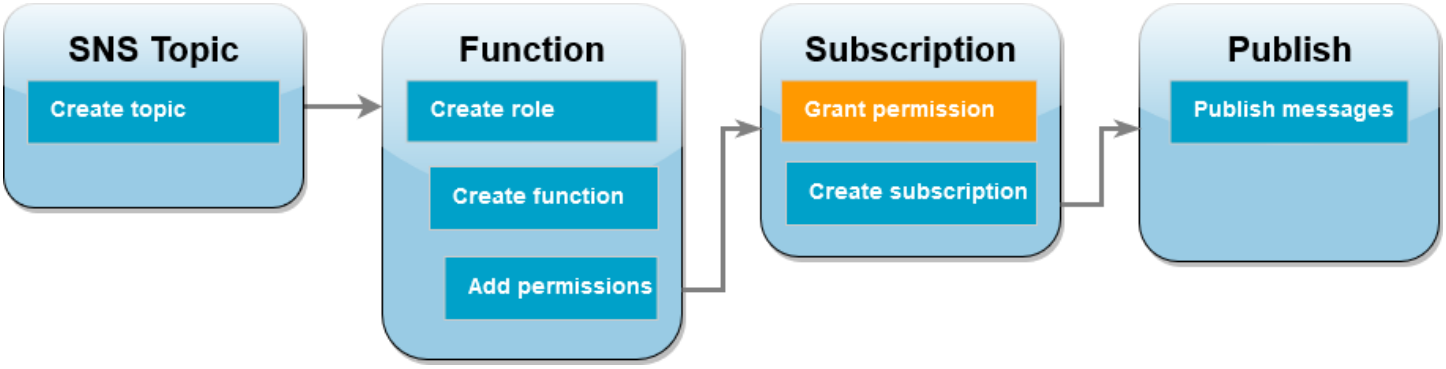
```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

  Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> Wenn das Konto mit dem Amazon SNS SNS-Thema in einem Opt-In gehostet wird AWS-Region, müssen Sie die Region im Prinzipal angeben. Wenn Sie beispielsweise mit einem Amazon-SNS-Thema in der Region Asien-Pazifik (Hongkong) arbeiten, muss für den Prinzipal `sns.ap-east-1.amazonaws.com` anstelle von `sns.amazonaws.com` angegeben werden.

Erteilen einer kontoübergreifenden Berechtigung für das Amazon-SNS-Abonnement (Konto A)



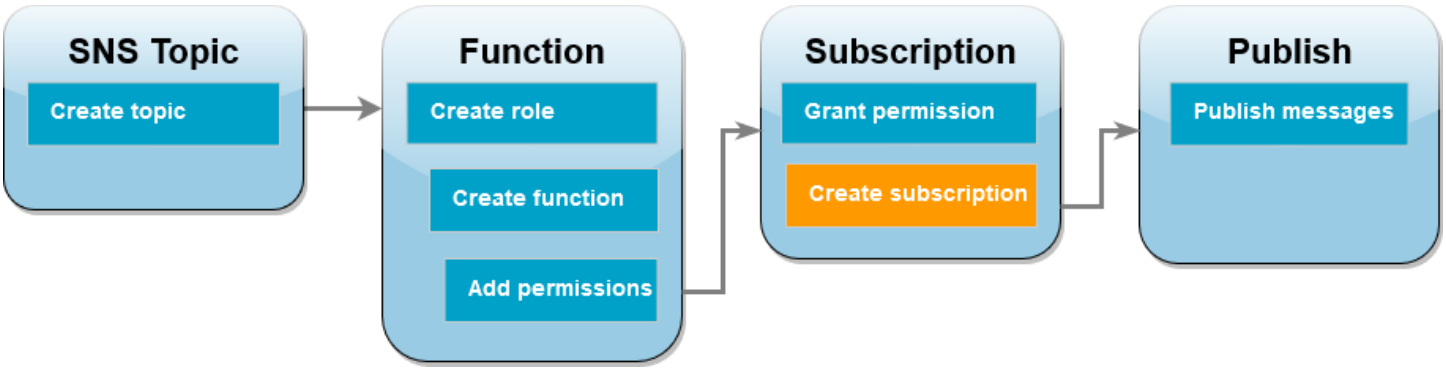
Damit Ihre Lambda-Event-Funktion in Konto B das Amazon SNS SNS-Thema abonnieren kann, das Sie in Konto A erstellt haben, müssen Sie Konto B die Erlaubnis erteilen, Ihr Thema zu abonnieren. Sie erteilen diese Berechtigung mit dem AWS CLI `add-permission` Befehl.

Erteilen der Berechtigung zum Abonnieren des Themas durch Konto B

- Führen Sie in Konto A den folgenden AWS CLI Befehl aus. Verwenden Sie den ARN für das Amazon-SNS-Thema, den Sie sich zuvor notiert haben.

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

Erstellen eines Abonnements (Konto B)



In Konto B abonnieren Sie jetzt Ihre Lambda-Ereignisfunktion für das Amazon SNS SNS-Thema, das Sie zu Beginn des Tutorials in Konto A erstellt haben. Wenn eine Nachricht an dieses Thema gesendet wird (**`sns-topic-for-lambda`**), ruft Amazon SNS Ihre Lambda-Ereignisfunktion **`Function-With-SNS`** in Konto B auf.

Erstellen eines Abonnements

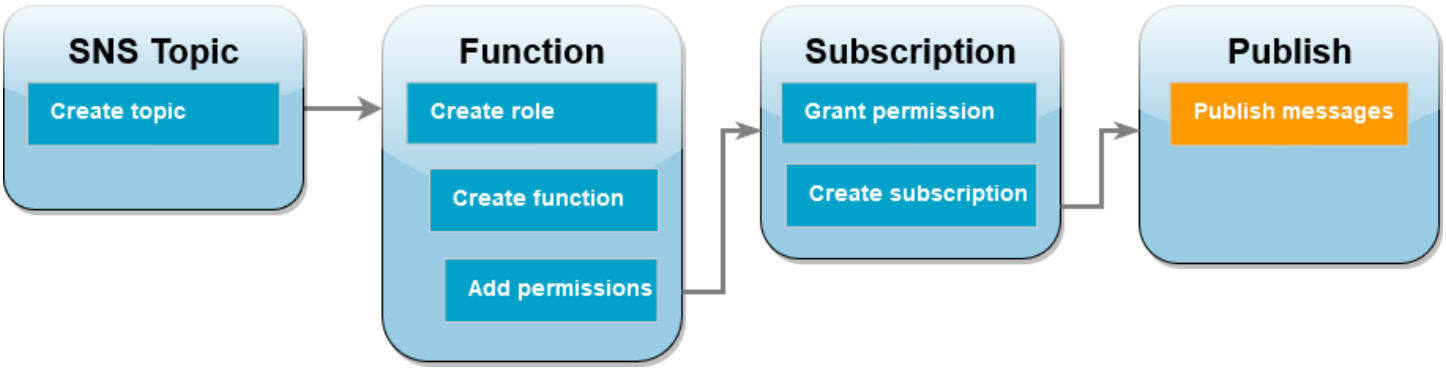
- Führen Sie in Konto B den folgenden Befehl aus. AWS CLI Verwenden Sie Ihre Standardregion, in der Sie Ihr Thema erstellt haben, und die ARNs für Ihr Thema und die Lambda Event Function.

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

Veröffentlichen von Nachrichten für das Thema (Konto A und Konto B)



Jetzt, da Ihre Lambda-Event-Funktion in Konto B Ihr Amazon SNS SNS-Thema in Konto A abonniert hat, ist es an der Zeit, Ihre Einrichtung zu testen, indem Sie Nachrichten zu Ihrem Thema veröffentlichen. Um zu bestätigen, dass Amazon SNS Ihre Lambda-Ereignisfunktion aufgerufen hat, verwenden Sie CloudWatch Logs, um die Ausgabe Ihrer Funktion einzusehen.

Um eine Nachricht zu Ihrem Thema zu veröffentlichen und die Ausgabe Ihrer Event-Funktion anzusehen

1. Geben Sie `Hello World` in eine Textdatei ein und speichern Sie sie als `message.txt`.

2. Führen Sie aus demselben Verzeichnis, in dem Sie Ihre Textdatei gespeichert haben, den folgenden AWS CLI Befehl in Konto A aus. Verwenden Sie den ARN für Ihr eigenes Thema.

   ```
   aws sns publish --message file://message.txt --subject Test \
       --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
       --profile accountA
   ```

   Dadurch wird eine Nachrichten-ID mit einer eindeutigen Kennung zurückgegeben, die angibt, dass Amazon SNS die Nachricht akzeptiert hat. Amazon SNS versucht dann, die Nachricht an die Abonnenten des Themas weiterzuleiten. Um zu bestätigen, dass Amazon SNS Ihre Lambda-Ereignisfunktion aufgerufen hat, verwenden Sie CloudWatch Logs, um die Ausgabe Ihrer Funktion einzusehen:
3. Öffnen Sie in Konto B die Seite Protokollgruppen der CloudWatch Amazon-Konsole.
4. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/Function-With-SNS`) aus.
5. Wählen Sie den neuesten Protokollstreams aus.
6. Wenn Ihre Funktion korrekt aufgerufen wurde, wird eine Ausgabe ähnlich der folgenden angezeigt, die den Inhalt der Nachricht zeigt, die Sie zu Ihrem Thema veröffentlicht haben.

   ```
   2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
    message Hello World
   2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
   ```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

Bereinigen Sie über Konto A Ihr Amazon-SNS-Thema.

So löschen Sie das Amazon-SNS-Thema

1. Öffnen Sie die Seite Themen der Amazon-SNS-Konsole.
2. Wählen Sie das Thema aus, das Sie gerade erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie **`delete me`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

Bereinigen Sie in Konto B Ihre Ausführungsrolle, Lambda Event Function und Ihr Amazon SNS SNS-Abonnement.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie das Amazon-SNS-Abonnement

1. Öffnen Sie die Seite Abonnements der Amazon-SNS-Konsole.
2. Wählen Sie das von Ihnen erstellte Abonnement aus.
3. Wählen Sie Löschen, Löschen aus.

Langlebige Lambda-Funktionen

Mit AWS Lambda Dank robuster Funktionen können Sie robuste mehrstufige Anwendungen und KI-Workflows erstellen, die bis zu einem Jahr lang ausgeführt werden können und trotz Unterbrechungen einen zuverlässigen Fortschritt gewährleisten. Wenn eine dauerhafte Funktion ausgeführt wird, wird dieser komplette Lebenszyklus als dauerhafte Ausführung bezeichnet. Dabei werden Checkpoints verwendet, um den Fortschritt zu verfolgen und Fehler automatisch durch Wiederholung zu beheben. Dabei wird die Ausführung von Anfang an wiederholt, während abgeschlossene Arbeiten übersprungen werden.

Innerhalb jeder Funktion verwenden Sie dauerhafte Operationen als grundlegende Bausteine. Schritte führen Geschäftslogik mit integrierten Wiederholungsversuchen und Fortschrittsverfolgung

aus, während Wartezeiten die Ausführung aussetzen, ohne dass Rechenkosten anfallen. Somit eignen sie sich ideal für lang andauernde Prozesse wie Human-in-the-Loop-Workflows oder das Abfragen externer Abhängigkeiten. Ganz gleich, ob Sie Aufträge verarbeiten, Microservices koordinieren oder agentische KI-Anwendungen orchestrieren — langlebige Funktionen behalten den Status automatisch bei und stellen sich nach Ausfällen wieder her, während Sie Code in vertrauten Programmiersprachen schreiben.

> **Note**
>
> Dieses Handbuch behandelt Lambda-Servicethemen wie Bereitstellung, Konfiguration, Berechtigungen, Aufruf, Überwachung und Kontingente. Referenzmaterial zur SDK-Programmierung, Schnellstart-Tutorials und sprachspezifische Anleitungen finden Sie im AWS Durable Execution SDK-Entwicklerhandbuch.

Wichtigste Vorteile

Schreiben Sie auf natürliche Weise belastbaren Code: Mit vertrauten Programmierkonstrukten schreiben Sie Code, der Fehler automatisch behandelt. Built-in Checkpointing, transparente Wiederholungsversuche und automatische Wiederherstellung sorgen dafür, dass Ihre Geschäftslogik klar und konzentriert bleibt.

Zahlen Sie nur für das, was Sie tatsächlich nutzen: Bei Wartevorgängen wird Ihre Funktion unterbrochen, ohne dass Rechenkosten anfallen. Bei Workflows mit langer Laufzeit, die Stunden oder Tage warten, zahlen Sie nur für die tatsächliche Verarbeitungszeit, nicht für Wartezeiten im Leerlauf.

Einfache Bedienung: Mit dem serverlosen Modell von Lambda erhalten Sie automatische Skalierung, einschließlich einer Skalierung auf Null, ohne die Infrastruktur verwalten zu müssen. Stabile Funktionen übernehmen das Zustandsmanagement, die Logik für Wiederholungsversuche und die Wiederherstellung nach einem Ausfall automatisch und reduzieren so den betrieblichen Aufwand.

Wann sollten langlebige Funktionen verwendet werden

Short-lived Koordination: Koordinieren Sie Zahlungen, Inventar und Versand für mehrere Services mit automatischem Rollback bei Ausfällen. Bearbeiten Sie Bestellungen von der Validierung über die Zahlungsautorisierung bis hin zur Inventarzuweisung und Erfüllung — mit garantierter Fertigstellung.

Verarbeiten Sie Zahlungen mit Zuversicht: Schaffen Sie belastbare Zahlungsströme, die den Transaktionsstatus auch bei Ausfällen aufrechterhalten und Wiederholungen automatisch

verarbeiten. Koordinieren Sie mehrstufige Autorisierungen, Betrugsprüfungen und Abwicklungen zwischen Zahlungsanbietern und sorgen Sie so für vollständige Überprüfbarkeit in allen Schritten.

Erstellen Sie zuverlässige KI-Workflows: Erstellen Sie mehrstufige KI-Workflows, die Modellabrufe verketten, menschliches Feedback einbeziehen und lang andauernde Aufgaben bei Ausfällen deterministisch behandeln. Wird nach einer Unterbrechung automatisch fortgesetzt und es wird nur für die aktive Ausführungszeit bezahlt.

Orchestrieren Sie die Erfüllung komplexer Bestellungen: Koordinieren Sie die Auftragsabwicklung über Bestands-, Zahlungs-, Versand- und Benachrichtigungssysteme hinweg mit integrierter Ausfallsicherheit. Behandeln Sie Teilausfälle automatisch, behalten Sie den Auftragsstatus trotz Unterbrechungen bei und warten Sie effizient auf externe Ereignisse, ohne Rechenressourcen zu verbrauchen.

Automatisieren Sie mehrstufige Geschäftsabläufe: Erstellen Sie zuverlässige Workflows für das Onboarding von Mitarbeitern, Kreditgenehmigungen und Compliance-Prozesse, die sich über Tage oder Wochen erstrecken. Behalten Sie den Status Ihrer Arbeitsabläufe bei Genehmigungen durch Mitarbeiter, Systemintegrationen und geplanten Aufgaben bei und bieten Sie gleichzeitig einen vollständigen Überblick über den Prozessstatus und die Historie.

Wie lassen sich langlebige Funktionen mit Step Functions vergleichen

Sowohl langlebige Funktionen als auch Step Functions ermöglichen eine Workflow-Orchestrierung mit automatischer Statusverwaltung. Die wichtigsten Unterschiede bestehen darin, wo sie ausgeführt werden und wie Sie Workflows definieren:

- Dauerhafte Funktionen: Wird innerhalb von Lambda ausgeführt, verwendet Standardprogrammiersprachen und wird in der Lambda-Umgebung verwaltet
- Step-Funktionen: Eigenständiger Service, grafischer DSL- oder Grafikdesigner, vollständig verwaltet und wartungsfrei

Langlebige Funktionen eignen sich ideal für die Anwendungsentwicklung in Lambda, bei der Arbeitsabläufe eng mit der Geschäftslogik verknüpft sind. Step Functions zeichnet sich durch die Orchestrierung von Workflows in allen Bereichen aus AWS Dienste, bei denen Sie visuelles Design, native Integrationen für über 220 Dienste und eine wartungsfreie Infrastruktur benötigen.

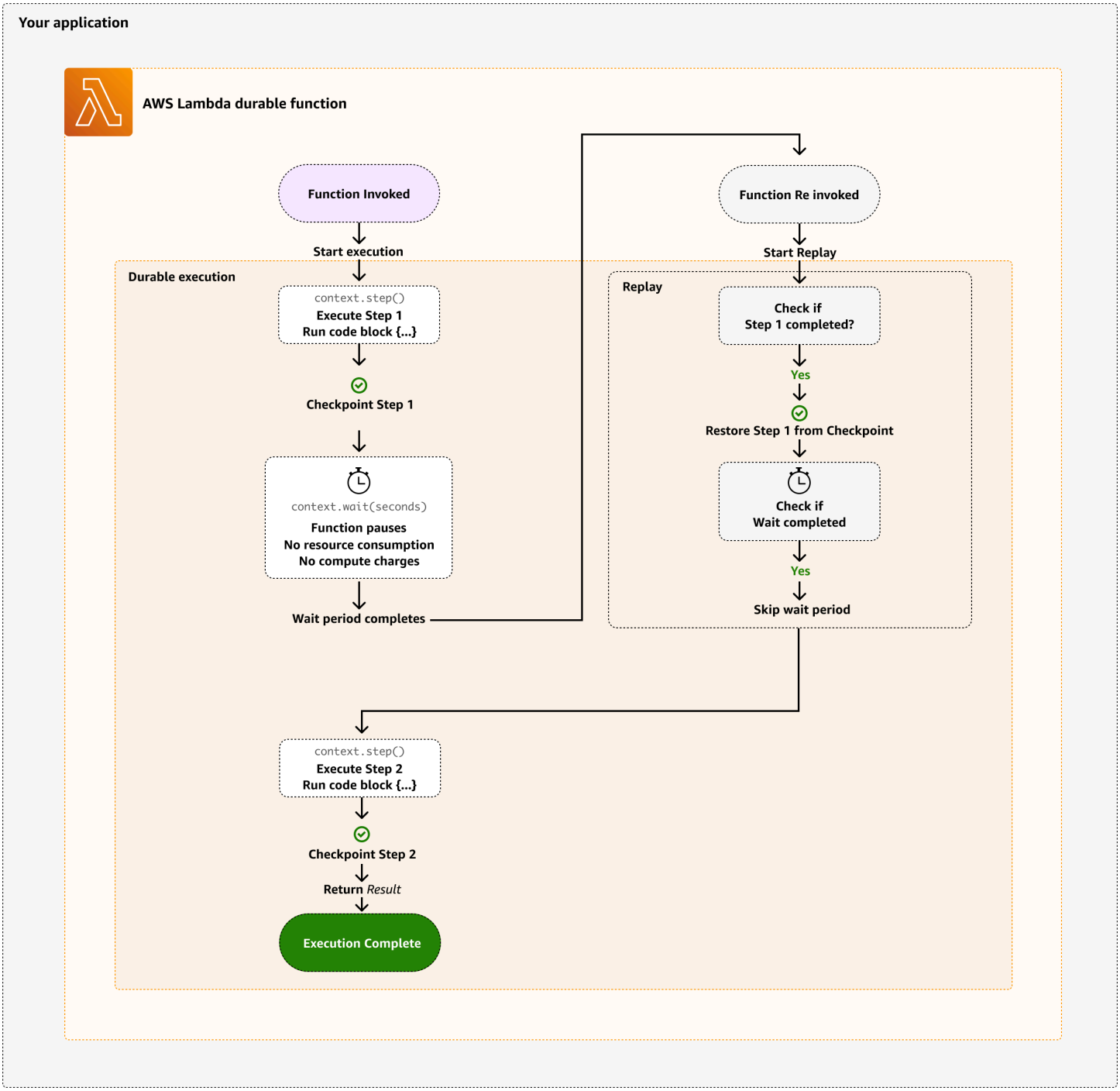
Einen ausführlichen Vergleich finden Sie unter Dauerhafte Funktionen oder Step-Funktionen.

Funktionsweise

Unter der Haube befinden sich langlebige Funktionen wie normale Lambda-Funktionen mit einem checkpoint/replay Mechanismus zur Überwachung des Fortschritts und zur Unterstützung lang andauernder Operationen durch benutzerdefinierte Haltepunkte, der allgemein als dauerhafte Ausführung bezeichnet wird. Wenn Ihre Funktion nach einer Pause oder Unterbrechung wieder aufgenommen wird, führt das System eine Wiederholung durch. Während der Wiedergabe wird Ihr Code von Anfang an ausgeführt, überspringt jedoch abgeschlossene Checkpoints und verwendet gespeicherte Ergebnisse, anstatt abgeschlossene Operationen erneut auszuführen. Dieser Wiedergabemechanismus gewährleistet Konsistenz und ermöglicht gleichzeitig lang andauernde Ausführungen.

Um diesen Checkpoint- und Replay-Mechanismus in Ihren Anwendungen zu nutzen, bietet Lambda ein robustes Ausführungs-SDK. Das SDK abstrahiert die Komplexität der Verwaltung von Checkpoints und Replay und macht einfache Primitive verfügbar, die als dauerhafte Operationen bezeichnet werden und die Sie in Ihrem Code verwenden. Das SDK ist verfügbar für JavaScript, TypeScript, Python, Java, Go und C# (.NET), die sich nahtlos in Ihren bestehenden Lambda-Entwicklungsworkflow integrieren lassen.

Mit dem SDK verpacken Sie Ihren Lambda-Event-Handler, der dann eine DurableContext neben Ihrer Veranstaltung. In diesem Kontext haben Sie Zugriff auf dauerhafte Operationen wie Schritte und Wartezeiten. Sie schreiben Ihre Funktionslogik als normalen sequentiellen Code, aber anstatt Dienste direkt aufzurufen, fassen Sie diese Aufrufe in Schritte ein, um automatische Checkpoints und Wiederholungen zu ermöglichen. Wenn Sie die Ausführung unterbrechen müssen, fügen Sie Wartezeiten hinzu, die Ihre Funktion aussetzen, ohne dass Gebühren anfallen. Das SDK bewältigt die gesamte Komplexität der Statusverwaltung und der Wiedergabe hinter den Kulissen, sodass Ihr Code sauber und lesbar bleibt.



Nächste Schritte

- AWS Durable Execution SDK — Entwicklerhandbuch — SDK-Referenz, Schnellstarts, Tests und Sprachleitfäden
- Beginnen Sie mit langlebigen Funktionen
- Langlebige Funktionen oder Step-Funktionen

- Überwachen und debuggen Sie langlebige Funktionen
- Überprüfen Sie die Sicherheit und die Berechtigungen
- Verschlüsseln AWS Lambda dauerhafte Ausführungsdaten
- Folgen Sie den bewährten Methoden

Grundkonzepte

Das AWS Das Durable Execution SDK ist die Grundlage für die Erstellung langlebiger Funktionen und bietet die Primitive, die Sie benötigen, um den Fortschritt zu überprüfen, Wiederholungsversuche zu verarbeiten und den Ausführungsablauf zu verwalten. Das SDK ist verfügbar für Node.js (JavaScript und TypeScript), Python, Java, Go und C# (.NET). Eine vollständige SDK-Dokumentation einschließlich API-Referenz, Schnellstart-Tutorials und sprachspezifischen Anleitungen finden Sie im AWS Durable Execution SDK — Entwicklerhandbuch.

Dauerhafte Ausführung

EIN dauerhafte Ausführung stellt den gesamten Lebenszyklus einer dauerhaften Lambda-Funktion dar und verwendet einen Checkpoint- und Wiedergabemechanismus, um den Fortschritt der Geschäftslogik zu verfolgen, die Ausführung zu unterbrechen und nach Ausfällen eine Wiederherstellung durchzuführen. Wenn Funktionen nach einer Unterbrechung oder Unterbrechung wieder aufgenommen werden, werden zuvor abgeschlossene Checkpoints erneut abgespielt und die Funktion setzt die Ausführung fort.

Der Lebenszyklus kann mehrere Aufrufe einer Lambda-Funktion umfassen, um die Ausführung abzuschließen, insbesondere nach Unterbrechungen oder einer Wiederherstellung nach einem Ausfall. Mit diesem Ansatz kann Ihre Funktion über einen längeren Zeitraum (bis zu einem Jahr) ausgeführt werden, während trotz Unterbrechungen ein zuverlässiger Ablauf gewährleistet ist.

Wie funktioniert die Wiederholung

Lambda führt ein laufendes Protokoll aller dauerhaften Operationen (Schritte, Wartezeiten und andere Operationen), während Ihre Funktion ausgeführt wird. Wenn Ihre Funktion unterbrochen werden muss oder auf eine Unterbrechung stößt, speichert Lambda dieses Checkpoint-Log und stoppt die Ausführung. Wenn es Zeit ist, die Funktion fortzusetzen, ruft Lambda Ihre Funktion erneut von Anfang an auf und spielt das Checkpoint-Log erneut ab, wobei abgeschlossene Operationen durch gespeicherte Werte ersetzt werden. Das bedeutet, dass Ihr Code erneut ausgeführt wird, zuvor abgeschlossene Schritte jedoch nicht erneut ausgeführt werden. Ihre gespeicherten Ergebnisse werden stattdessen verwendet.

Dieser Wiedergabemechanismus ist grundlegend für das Verständnis dauerhafter Funktionen. Ihr Code muss während der Wiedergabe deterministisch sein, was bedeutet, dass er bei gleichen Eingaben dieselben Ergebnisse liefert. Vermeiden Sie Operationen mit Nebeneffekten (wie das Generieren von Zufallszahlen oder das Abrufen der aktuellen Uhrzeit) außerhalb von Schritten, da diese beim Abspielen unterschiedliche Werte erzeugen und zu nicht deterministischem Verhalten führen können.

DurableContext

DurableContext ist das Kontextobjekt, das Ihre langlebige Funktion empfängt. Es bietet Methoden für dauerhafte Operationen wie Schritte und Wartezeiten, mit denen Checkpoints erstellt und der Ausführungsablauf verwaltet wird.

Ihre dauerhafte Funktion erhält eine `DurableContext` anstelle des Standard-Lambda-Kontexts:

TypeScript

```
import {
  DurableContext,
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const result = await context.step(async () => {
      return "step completed";
    });
    return result;
  },
);
```

Python

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)

@durable_step
def my_step(step_context, data):
```

```
    # Your business logic
    return result

@durable_execution
def handler(event, context: DurableContext):
    result = context.step(my_step(event["data"]))
    return result
```

Durch `DurableContext`, hat Ihre Funktion Zugriff auf dauerhafte Operationen, die den Fortschritt kontrollieren und die Arbeit koordinieren:

- Schritte Führen Sie eine Geschäftslogik mit integrierten Wiederholungsversuchen und automatischem Checkpointing aus. Jeder Schritt speichert sein Ergebnis, sodass Ihre Funktion nach einer Unterbrechung ab dem letzten abgeschlossenen Schritt wieder aufgenommen wird.
- Wartet sind geplante Pausen, in denen Ihre Funktion nicht mehr läuft und der Ladevorgang unterbrochen wird, bis es Zeit ist, fortzufahren. Verwenden Sie sie für Zeiträume, externe Rückrufe oder zum Abfragen einer Bedingung.
- Aufrufen ruft andere Lambda-Funktionen auf, einschließlich langlebiger Funktionen, und überprüft das Ergebnis. Cross-account Aufrufe werden nicht unterstützt.

Die vollständige Sprachreferenz zu allen verfügbaren dauerhaften Operationen finden Sie unter SDK-Referenz in der AWS Durable Execution SDK-Entwicklerhandbuch.

Dauerhafte Funktionskonfiguration

Dauerhafte Funktionen fügen Konfigurationseinstellungen hinzu, die das Ausführungsverhalten und die Datenspeicherung steuern. Diese Einstellungen gelten zusätzlich zur vorhandenen Konfiguration der Funktion, wie z. B. Speicher und Timeout für die Funktion. Sie gelten für die gesamte dauerhafte Ausführung und nicht für einzelne Aufrufe.

Das DurableConfig Objekt definiert die Konfiguration für langlebige Funktionen:

```
{
  "ExecutionTimeout": Integer,
  "RetentionPeriodInDays": Integer
}
```

Ausführungstimeout

Das Timeout bei der Ausführung steuert, wie lange eine dauerhafte Ausführung von Anfang bis Ende ausgeführt werden kann. Dies unterscheidet sich vom Timeout der Lambda-Funktion, der steuert, wie lange ein einzelner Funktionsaufruf ausgeführt werden kann.

Eine dauerhafte Ausführung kann sich über mehrere Aufrufe der Lambda-Funktion erstrecken, während sie Checkpoints, Wartezeiten und Wiederholungen durchläuft. Das Ausführungs-Timeout bezieht sich auf die gesamte verstrichene Zeit der dauerhaften Ausführung, nicht auf einzelne Funktionsaufrufen.

Den Unterschied verstehen

Das Funktions-Timeout begrenzt jeden einzelnen Aufruf Ihrer Funktion auf standardmäßig bis zu 15 Minuten oder 90 Minuten bei asynchronen Aufrufen und Aufrufen der Ereignisquellenzuordnung auf Lambda Managed Instances. Das dauerhafte Ausführungs-Timeout (maximal 1 Jahr) begrenzt die Gesamtzeit vom Beginn der Ausführung bis zum Abschluss, Fehlschlag oder Timeout. Während dieses Zeitraums wird Ihre Funktion möglicherweise mehrfach aufgerufen, da sie Schritte verarbeitet, wartet und Fehler wiederherstellt.

Wenn Sie beispielsweise ein dauerhaftes Ausführungs-Timeout von 24 Stunden und ein Timeout für die Lambda-Funktion von 5 Minuten festlegen:

- Jeder Funktionsaufruf muss innerhalb von 5 Minuten abgeschlossen sein
- Die gesamte dauerhafte Ausführung kann bis zu 24 Stunden dauern
- Ihre Funktion kann während dieser 24 Stunden viele Male aufgerufen werden
- Wartevorgänge werden nicht auf den Timeout der Lambda-Funktion angerechnet, wohl aber auf den Timeout für die Ausführung

Sie können das Ausführungs-Timeout konfigurieren, wenn Sie eine dauerhafte Funktion mithilfe der Lambda-Konsole erstellen. AWS CLI, oder AWS SAM. Wählen Sie in der Lambda-Konsole Ihre Funktion und dann Konfiguration, Dauerhafte Ausführung aus. Stellen Sie den Timeout-Wert für die Ausführung in Sekunden ein (Standard: 86400 Sekunden/24 Stunden, Minimum: 60 Sekunden, Maximum: 31536000 Sekunden/1 Jahr).

> **Note**
>
> Das Ausführungs-Timeout und das Funktions-Timeout sind unterschiedliche Einstellungen. Das Funktions-Timeout begrenzt jeden Aufruf, standardmäßig auf bis zu 15 Minuten oder 90

Minuten bei asynchronen Aufrufen und Aufrufen der Ereignisquellenzuordnung auf Lambda Managed Instances. Das Ausführungs-Timeout begrenzt die Gesamtzeit der dauerhaften Ausführung über alle ihre Aufrufe hinweg (maximal 1 Jahr).

Aufbewahrungszeitraum

Der Aufbewahrungsfrist steuert, wie lange Lambda den Ausführungsverlauf und die Checkpoint-Daten nach Abschluss einer dauerhaften Ausführung aufbewahrt. Zu diesen Daten gehören die Schrittergebnisse, der Ausführungsstatus und das vollständige Checkpoint-Protokoll.

Nach Ablauf der Aufbewahrungsfrist löscht Lambda den Ausführungsverlauf und die Checkpoint-Daten. Sie können keine Ausführungsdetails mehr abrufen oder die Ausführung wiederholen. Die Aufbewahrungsfrist beginnt, wenn die Ausführung einen Endzustand erreicht (SUCCEEDED, FAILED, STOPPED oder TIMED_OUT).

Sie können den Aufbewahrungszeitraum konfigurieren, wenn Sie eine dauerhafte Funktion mithilfe der Lambda-Konsole erstellen. AWS CLI, oder AWS SAM. Wählen Sie in der Lambda-Konsole Ihre Funktion und dann Konfiguration, Dauerhafte Ausführung aus. Legen Sie den Wert für die Aufbewahrungsdauer in Tagen fest (Standard: 14 Tage, Minimum: 1 Tag, Maximum: 90 Tage).

Wählen Sie einen Aufbewahrungszeitraum, der auf Ihren Compliance-Anforderungen, Debugging-Anforderungen und Kostenüberlegungen basiert. Längere Aufbewahrungsfristen bieten mehr Zeit für Debugging und Audits, erhöhen jedoch die Speicherkosten.

Weitere Informationen finden Sie auch unter

- AWS Durable Execution SDK — Entwicklerhandbuch — Vollständige SDK-Referenz, Schnellstart-Tutorials, Testframework und sprachspezifische Anleitungen.
- Dauerhafte Funktionen oder Step-Funktionen — Vergleichen Sie langlebige Funktionen mit Step Functions, um zu verstehen, wann jeder Ansatz am effektivsten ist.

Dauerhafte Lambda-Funktionen erstellen

Um mit dauerhaften Lambda-Funktionen zu beginnen, verwenden Sie die Lambda-Konsole, um eine dauerhafte Funktion zu erstellen. In wenigen Minuten können Sie eine dauerhafte Funktion erstellen und bereitstellen, die anhand von Schritten und Wartezeiten die Ausführung anhand von Checkpoints demonstriert.

Sie können auch dauerhafte Funktionen mithilfe von Infrastructure as Code (AWS CLIIaC) erstellen.

Während der Durchführung des Tutorials lernen Sie grundlegende Konzepte für dauerhafte Funktionen kennen, z. B. wie Sie das `DurableContext` Objekt verwenden, Checkpoints mit Schritten erstellen und die Ausführung mit Wartezeiten unterbrechen. Außerdem erfahren Sie, wie die Wiedergabe funktioniert, wenn Ihre Funktion nach einer Wartezeit wieder aufgenommen wird.

Um die Dinge einfach zu halten, zeigt Ihnen dieses Tutorial, wie Sie Ihre Funktion entweder mit Python oder Node.js Runtime erstellen. Mit diesen interpretierten Sprachen können Sie Funktionscode direkt im integrierten Code-Editor der Konsole bearbeiten.

> **Note**
>
> Dauerhafte Funktionen unterstützen derzeit Python-, Node.js (JavaScript/TypeScript)- und Java-Laufzeiten sowie Container-Images (OCI). Eine vollständige Liste der unterstützten Laufzeitversionen und Container-Image-Optionen finden Sie unter Unterstützte Laufzeiten für langlebige Funktionen. Weitere Informationen zur Verwendung von Container-Images mit Lambda finden Sie unter Creating Lambda container images im Lambda Developer Guide.

> **Tip**
>
> Weitere Informationen zum Erstellen von Serverless-Lösungen finden Sie im Serverless-Benutzerhandbuch.

Erstellen Sie mit der Konsole eine dauerhafte Lambda-Funktion

In diesem Beispiel verarbeitet Ihre langlebige Funktion eine Bestellung in mehreren Schritten mit automatischem Checkpoint. Die Funktion verwendet ein JSON-Objekt, das eine Bestell-ID enthält, validiert die Bestellung, verarbeitet die Zahlung und bestätigt die Bestellung. Jeder Schritt wird automatisch überprüft. Wenn die Funktion unterbrochen wird, wird sie mit dem letzten abgeschlossenen Schritt fortgesetzt.

Ihre Funktion demonstriert auch eine Warteoperation, bei der die Ausführung für einen kurzen Zeitraum angehalten wird, um das Warten auf eine externe Bestätigung zu simulieren.

Um eine dauerhafte Funktion mit der Konsole zu erstellen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.

2. Wählen Sie „Event-Funktion erstellen“.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie im Bereich Grundlegende Informationen als Funktionsname **`myDurableFunction`** ein.
5. Wählen Sie für Runtime entweder Node.js 24 oder Python 3.14.
6. Wählen Sie Dauerhafte Ausführung aktivieren aus.

Lambda erstellt Ihre dauerhafte Funktion mit einer Ausführungsrolle, die Berechtigungen für Checkpoint-Operationen (`lambda:CheckpointDurableExecution`und`lambda:GetDurableExecutionState`) beinhaltet.

> **Note**
>
> Lambda-Laufzeiten enthalten das Durable Execution SDK, sodass Sie langlebige Funktionen ohne Paketierungsabhängigkeiten testen können. Wir empfehlen jedoch, das SDK in Ihr Bereitstellungspaket für die Produktion aufzunehmen. Dadurch wird die Versionskonsistenz gewährleistet und potenzielle Runtime-Updates vermieden, die Ihre Funktion beeinträchtigen könnten.

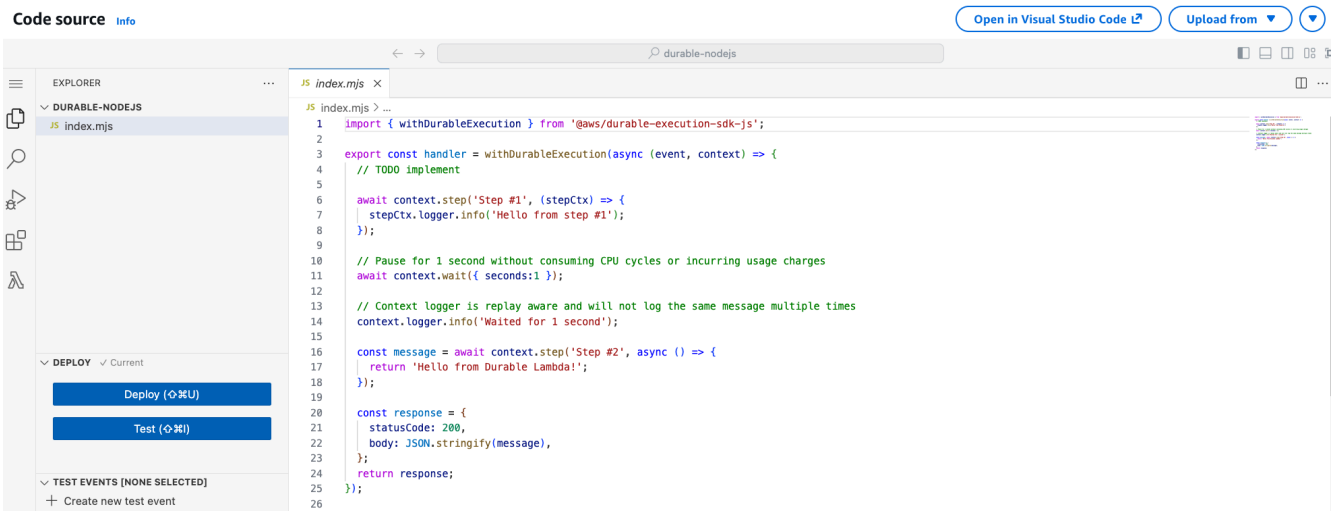
Verwenden Sie den integrierten Code-Editor der Konsole, um Ihren dauerhaften Funktionscode hinzuzufügen.

Node.js

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte index.mjs im Code-Editor nicht angezeigt wird, wählen Sie index.mjs im Datei-Explorer aus, wie im folgenden Diagramm gezeigt.

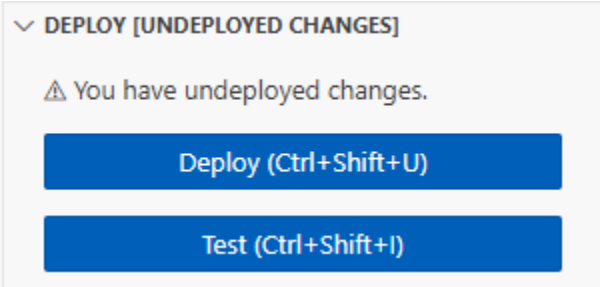


2. Fügen Sie den folgenden Code in die Registerkarte index.mjs ein und ersetzen Sie den von Lambda erstellten Code.

```
import {
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event, context) => {
    const orderId = event.orderId;

    // Step 1: Validate order
    const validationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Validating order ${orderId}`);
      return { orderId, status: "validated" };
    });

    // Step 2: Process payment
    const paymentResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Processing payment for order ${orderId}`);
      return { orderId, status: "paid", amount: 99.99 };
    });

    // Wait for 10 seconds to simulate external confirmation
    await context.wait({ seconds: 10 });

    // Step 3: Confirm order
    const confirmationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Confirming order ${orderId}`);
      return { orderId, status: "confirmed" };
```

```
    });

    return {
      orderId: orderId,
      status: "completed",
      steps: [validationResult, paymentResult, confirmationResult]
    };
  }
);
```

3. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Verstehen Sie Ihren dauerhaften Funktionscode

Bevor Sie mit dem nächsten Schritt fortfahren, sollten wir uns den Funktionscode ansehen und die wichtigsten Konzepte für langlebige Funktionen verstehen.

- Der `withDurableExecution` Wrapper:

  Ihre langlebige Funktion ist umhüllt von. `withDurableExecution` Dieser Wrapper ermöglicht eine dauerhafte Ausführung, indem er das `DurableContext` Objekt bereitstellt und Checkpoint-Operationen verwaltet.

- Das `DurableContext` Objekt:

  Anstelle des Standard-Lambda-Kontextes erhält Ihre Funktion eine`DurableContext`. Dieses Objekt bietet Methoden für dauerhafte Operationen wie `step()` und `wait()` die Erstellung von Checkpoints.

- Schritte und Checkpoints:

  Jeder `context.step()` Aufruf erstellt vor und nach der Ausführung einen Checkpoint. Wenn Ihre Funktion unterbrochen wird, wird sie ab dem letzten abgeschlossenen Checkpoint wieder

aufgenommen. Die Funktion führt abgeschlossene Schritte nicht erneut aus. Stattdessen werden ihre gespeicherten Ergebnisse verwendet.

- Warten Sie auf Operationen:

  Der `context.wait()` Aufruf unterbricht die Ausführung, ohne Rechenressourcen zu verbrauchen. Wenn das Warten abgeschlossen ist, ruft Lambda Ihre Funktion erneut auf und spielt das Checkpoint-Protokoll erneut ab, wobei abgeschlossene Schritte durch gespeicherte Werte ersetzt werden.

- Wiedergabemechanismus:

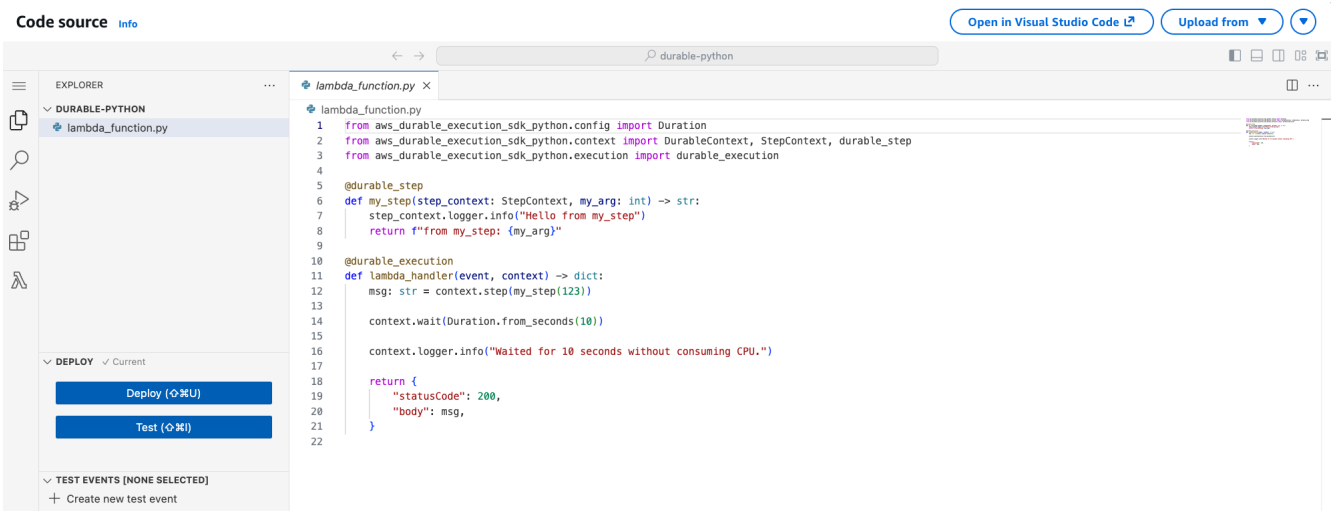
  Wenn Ihre Funktion nach einer Wartezeit oder Unterbrechung wieder aufgenommen wird, führt Lambda Ihren Code von Anfang an aus. Abgeschlossene Schritte werden jedoch nicht erneut ausgeführt. Lambda gibt ihre Ergebnisse aus dem Checkpoint-Log wieder. Aus diesem Grund muss Ihr Code deterministisch sein.

Python

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte lambda_function.py im Code-Editor nicht angezeigt wird, wählen Sie lambda_function.py im Datei-Explorer aus, wie im folgenden Diagramm dargestellt.

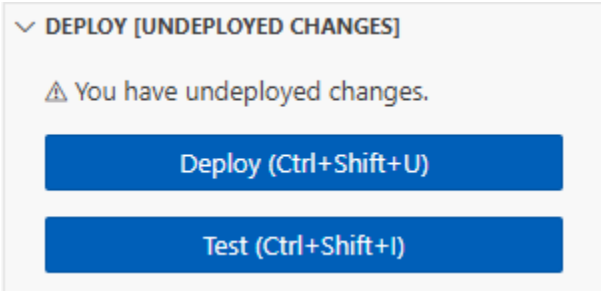


2. Fügen Sie den folgenden Code in die Registerkarte lambda_function.py ein und ersetzen Sie den von Lambda erstellten Code.

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)
from aws_durable_execution_sdk_python.config import Duration

@durable_step
def validate_order(step_context, order_id):
    step_context.logger.info(f"Validating order {order_id}")
    return {"orderId": order_id, "status": "validated"}

@durable_step
def process_payment(step_context, order_id):
    step_context.logger.info(f"Processing payment for order {order_id}")
    return {"orderId": order_id, "status": "paid", "amount": 99.99}

@durable_step
def confirm_order(step_context, order_id):
    step_context.logger.info(f"Confirming order {order_id}")
    return {"orderId": order_id, "status": "confirmed"}

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']

    # Step 1: Validate order
    validation_result = context.step(validate_order(order_id))

    # Step 2: Process payment
    payment_result = context.step(process_payment(order_id))

    # Wait for 10 seconds to simulate external confirmation
    context.wait(Duration.from_seconds(10))

    # Step 3: Confirm order
    confirmation_result = context.step(confirm_order(order_id))

    return {
        "orderId": order_id,
```

```
        "status": "completed",
        "steps": [validation_result, payment_result, confirmation_result]
    }
```

3. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Verstehen Sie Ihren dauerhaften Funktionscode

Bevor Sie mit dem nächsten Schritt fortfahren, sollten wir uns den Funktionscode ansehen und die wichtigsten Konzepte für langlebige Funktionen verstehen.

- Der `@durable_execution` Dekorateur:

  Ihre Handler-Funktion ist mit `@durable_execution` dekoriert. Dieser Decorator ermöglicht eine dauerhafte Ausführung, indem er das `DurableContext` Objekt bereitstellt und Checkpoint-Operationen verwaltet.
- Der `@durable_step` Dekorateur:

  Jede Schrittfunktion ist mit `@durable_step` dekoriert. Dieser Dekorateur kennzeichnet die Funktion als dauerhaften Schritt, der Checkpoints erzeugt.
- Das `DurableContext` Objekt:

  Anstelle des Standard-Lambda-Kontextes erhält Ihre Funktion eine`DurableContext`. Dieses Objekt bietet Methoden für dauerhafte Operationen wie `step()` und `wait()` die Erstellung von Checkpoints.
- Schritte und Checkpoints:

  Jeder `context.step()` Aufruf erstellt vor und nach der Ausführung einen Checkpoint. Wenn Ihre Funktion unterbrochen wird, wird sie ab dem letzten abgeschlossenen Checkpoint wieder aufgenommen. Die Funktion führt abgeschlossene Schritte nicht erneut aus. Stattdessen werden ihre gespeicherten Ergebnisse verwendet.

- Warten Sie auf Operationen:

  Der `context.wait()` Aufruf unterbricht die Ausführung, ohne Rechenressourcen zu verbrauchen. Wenn das Warten abgeschlossen ist, ruft Lambda Ihre Funktion erneut auf und spielt das Checkpoint-Protokoll erneut ab, wobei abgeschlossene Schritte durch gespeicherte Werte ersetzt werden.

- Das Python-SDK ist synchron:

  Beachten Sie, dass das Python-SDK nicht verwendet`await`. Alle dauerhaften Operationen sind synchrone Methodenaufrufen.

Rufen Sie die dauerhafte Funktion mit dem Code-Editor der Konsole auf

Wenn keine explizite Version angegeben (oder veröffentlicht) ist, ruft die Konsole die dauerhafte Funktion mithilfe des `$LATEST` Versionskennzeichners auf. Für die deterministische Ausführung Ihres Codes müssen Sie jedoch immer einen qualifizierten ARN verwenden, der auf eine stabile Version verweist.

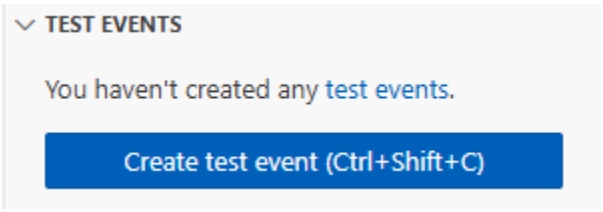
Um eine Version Ihrer Funktion zu veröffentlichen

1. Wählen Sie den Tab Versionen.
2. Wählen Sie Publish new version (Neue Version veröffentlichen) aus.
3. Geben Sie als Versionsbeschreibung **`Initial version`** (optional) ein.
4. Wählen Sie Publish.
5. Lambda erstellt Version 1 Ihrer Funktion. Beachten Sie, dass die Funktion ARN jetzt `:1` am Ende steht, was darauf hinweist, dass es sich um Version 1 handelt.

Erstellen Sie nun ein Testereignis, das an Ihre Funktion gesendet werden soll. Das Ereignis ist ein Dokument im JSON-Format, das eine Bestell-ID enthält.

So erstellen Sie das Testereignis

1. Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolen-Code-Editors die Option Testereignis erstellen aus.



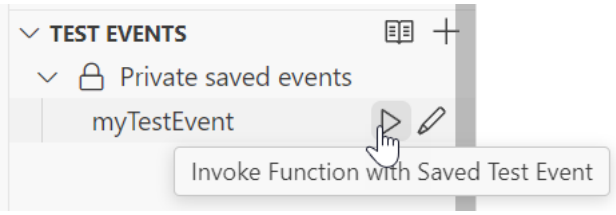
2. Geben Sie als Event name (Ereignisname) die Zeichenfolge **`myTestEvent`** ein.
3. Ersetzen Sie im Abschnitt Event JSON das Standard-JSON durch Folgendes:

```
{
  "orderId": "order-12345"
}
```

4. Wählen Sie Speichern.

Um Ihre dauerhafte Funktion zu testen und die Ausführung anzusehen

Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolencode-Editors das Ausführungssymbol neben Ihrem Testereignis:



Ihre dauerhafte Funktion wird ausgeführt. Da sie eine Wartezeit von 10 Sekunden beinhaltet, wird der erste Aufruf schnell abgeschlossen, und die Funktion wird nach Ablauf der Wartezeit wieder aufgenommen. Sie können den Ausführungsfortschritt auf der Registerkarte Dauerhafte Ausführungen einsehen.

Um die Ausführung Ihrer dauerhaften Funktion einzusehen

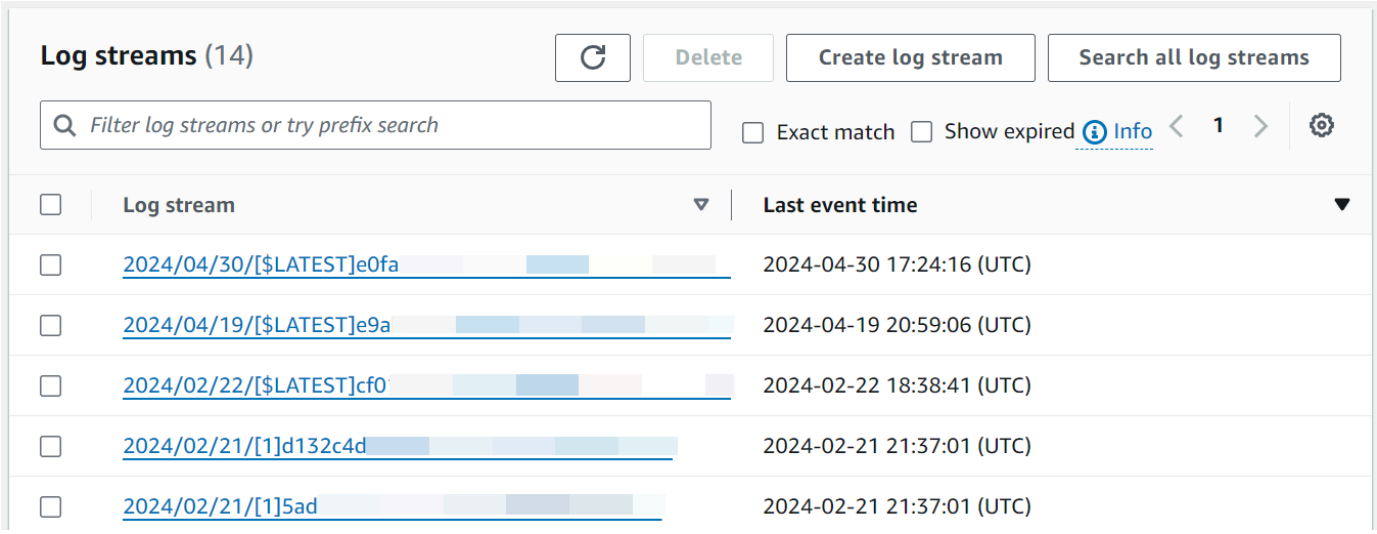
1. Wählen Sie die Registerkarte Dauerhafte Ausführungen.
2. Suchen Sie in der Liste nach Ihrer Hinrichtung. Die Ausführung zeigt den aktuellen Status an (Wird ausgeführt, Erfolgreich oder Fehlgeschlagen).
3. Wählen Sie die Ausführungs-ID, um Details anzuzeigen, darunter:
   - Ausführungszeitplan, der anzeigt, wann jeder Schritt abgeschlossen wurde
   - Verlauf des Checkpoints

- Wartezeiten
- Ergebnisse des Schritts

Sie können die Protokolle Ihrer Funktion auch unter Logs einsehen, um die Konsolenausgabe der einzelnen Schritte zu sehen. CloudWatch

Um die Aufrufaufzeichnungen Ihrer Funktion in CloudWatch Logs einzusehen

1. Öffnen Sie die Seite Protokollgruppen der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/myDurableFunction`) aus.
3. Scrollen Sie nach unten und wählen Sie den Protokollstream für die Funktionsaufrufen aus, die Sie sich ansehen möchten.



Sie sollten Protokolleinträge für jeden Aufruf Ihrer Funktion sehen, einschließlich der ersten Ausführung und der Wiederholung nach dem Warten.

> **Note**
>
> Wenn Sie den Logger über `DurableContext` (z. B. `context.logger` oder`stepContext.logger`) verwenden, werden Protokolle auch in den dauerhaften Ausführungs- und Schrittansichten der Lambda-Konsole angezeigt. Das Laden dieser Protokolle kann einen Moment dauern.

Bereinigen

Wenn Sie mit der dauerhaften Beispielfunktion fertig sind, löschen Sie sie. Sie können auch die Protokollgruppe löschen, in der die Protokolle der Funktion gespeichert sind, sowie die von der Konsole erstellte Ausführungsrolle.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Protokollgruppe

1. Öffnen Sie die Seite Log Groups (Protokollgruppen) der CloudWatch-Konsole.
2. Wählen Sie die Protokollgruppe der Funktion (`/aws/lambda/myDurableFunction`).
3. Wählen Sie Actions (Aktionen), Delete log group(s) (Protokollgruppe(n) löschen) aus.
4. Wählen Sie im Dialogfeld Delete log group(s) (Protokollgruppe(n) löschen) die Option Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Rollen der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie die Ausführungsrolle der Funktion aus (zum Beispiel `myDurableFunction-role-`*`31exxmpl`*).
3. Wählen Sie Löschen aus.
4. Geben Sie im Dialogfenster Delete role (Rolle löschen) den Namen der Rolle ein und wählen Sie anschließend Delete (Löschen) aus.

Zusätzliche Ressourcen und nächste Schritte

Nachdem Sie mit der Konsole eine einfache, dauerhafte Funktion erstellt und getestet haben, gehen Sie wie folgt vor:

- Erfahren Sie mehr über gängige Anwendungsfälle für langlebige Funktionen, darunter verteilte Transaktionen, Auftragsabwicklung und Workflows zur Überprüfung durch Mitarbeiter. Sehen Sie sich Beispiele an.
- Erfahren Sie, wie Sie die Ausführung dauerhafter Funktionen anhand von CloudWatch Metriken und Ausführungshistorie überwachen können. Weitere Informationen finden Sie unter Überwachung und Debuggen.
- Erfahren Sie, wie Sie langlebige Funktionen synchron und asynchron aufrufen und Ausführungen mit langer Laufzeit verwalten. Weitere Informationen finden Sie unter Dauerhafte Funktionen aufrufen.
- Folgen Sie den bewährten Methoden zum Schreiben von deterministischem Code, zur Verwaltung von Checkpoint-Größen und zur Kostenoptimierung. Siehe Bewährte Methoden.
- Erfahren Sie, wie Sie langlebige Funktionen lokal und in der Cloud testen können. Weitere Informationen finden Sie unter Testen langlebiger Funktionen.
- Vergleichen Sie langlebige Funktionen mit Step Functions, um herauszufinden, wann die einzelnen Methoden am effektivsten sind. Siehe Dauerhafte Funktionen oder Step Functions.

Stellen Sie langlebige Lambda-Funktionen bereit und rufen Sie sie auf mit dem AWS CLI

Verwenden Sie die AWS CLI , um dauerhafte Lambda-Funktionen mit imperativen Befehlen zu erstellen und bereitzustellen. Mit diesem Ansatz haben Sie die direkte Kontrolle über jeden Schritt des Bereitstellungsprozesses.

Voraussetzungen

- Installieren und Konfigurieren der AWS CLI. Anweisungen finden Sie unter Installation von AWS CLI.
- Erstellen Sie ein Bereitstellungspaket mit Ihrem Funktionscode und dem Durable Execution SDK.
- Erstellen Sie eine IAM-Ausführungsrolle mit Checkpoint-Berechtigungen.

Erstellen der Ausführungsrolle

Erstellen Sie eine IAM-Rolle mit Berechtigungen für grundlegende Lambda-Ausführungs- und Checkpoint-Operationen.

So erstellen Sie eine Ausführungsrolle

1. Erstellen Sie ein Dokument mit Vertrauensrichtlinien, das es Lambda ermöglicht, die Rolle zu übernehmen. Speichern Sie das unter: `trust-policy.json`

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. Erstellen Sie die Rolle:

```
aws iam create-role \
  --role-name durable-function-role \
  --assume-role-policy-document file://trust-policy.json
```

3. Fügen Sie die dauerhafte Ausführungsrichtlinie für Checkpoint-Operationen und die grundlegende Ausführung bei:

```
aws iam attach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy
```

Die `AWSLambdaBasicDurableExecutionRolePolicy` verwaltete Richtlinie umfasst die erforderlichen Berechtigungen für Checkpoint-Operationen (`lambda:CheckpointDurableExecution`und`lambda:GetDurableExecutionState`) und die grundlegende Lambda-Ausführung.

Erstellen Sie die dauerhafte Funktion

Erstellen Sie Ihre dauerhafte Funktion mit dem `--durable-config` Parameter.

Um eine dauerhafte Funktion zu erstellen

1. Packen Sie Ihren Funktionscode mit Abhängigkeiten in eine ZIP-Datei:

```
zip -r function.zip index.mjs node_modules/
```

> **Note**
>
> Für Java-basierte dauerhafte Funktionen müssen Sie Ihren Funktionscode und die Abhängigkeiten in einer einzigen ZIP-Datei oder Java-Archivdatei (JAR) kompilieren. Weitere Informationen finden Sie unter Bereitstellen von Java-Lambda-Funktionen mit ZIP- oder JAR-Dateiarchiven.

2. Erstellen Sie die Funktion mit aktivierter dauerhafter Ausführung:

```
aws lambda create-function \
  --function-name myDurableFunction \
  --runtime nodejs22.x \
  --role arn:aws:iam::123456789012:role/durable-function-role \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'
```

> **Note**
>
> Sie können die dauerhafte Ausführung nur aktivieren, wenn Sie die Funktion erstellen. Sie können es nicht für bestehende Funktionen aktivieren.

Veröffentlichen Sie eine Version

Dauerhafte Funktionen können zwar mit dem $LATEST Versionskennzeichner aufgerufen werden, Sie müssen jedoch immer einen qualifizierten ARN verwenden, der auf eine stabile Version verweist, um eine deterministische Ausführung Ihres Codes sicherzustellen.

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Initial version"
```

Der Befehl gibt die Version ARN zurück. Notieren Sie sich die Versionsnummer (z. B.`:1`) am Ende des ARN.

Erstellen Sie optional einen Alias, der auf die Version verweist:

```
aws lambda create-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 1
```

Rufen Sie die dauerhafte Funktion auf

Rufen Sie Ihre dauerhafte Funktion mit dem qualifizierten ARN (Version oder Alias) auf.

> **Note**
>
> Idempotente Aufrufe: Um doppelte Ausführungen bei wiederholten fehlgeschlagenen Aufrufen zu verhindern, können Sie einen Ausführungsnamen angeben, der die Ausführungssemantik sicherstellt. at-most-once Einzelheiten finden Sie unter Idempotenz.

Synchroner Aufruf

Verwenden Sie für Ausführungen, die innerhalb von 15 Minuten abgeschlossen sind, den synchronen Aufruf:

```
aws lambda invoke \
  --function-name myDurableFunction:1 \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Oder mit einem Alias:

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Asynchroner Aufruf

Verwenden Sie für lang andauernde Ausführungen den asynchronen Aufruf:

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --invocation-type Event \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Bei asynchronem Aufruf kehrt Lambda sofort zurück. Die Funktion wird weiterhin im Hintergrund ausgeführt.

> **Note**
>
> Sie können sie $LATEST für Prototyping und Tests in der Konsole verwenden. Verwenden Sie für Produktionsworkloads eine veröffentlichte Version oder einen Alias.

Verwalten Sie langlebige Ausführungen

Verwenden Sie die folgenden Befehle, um die Ausführung dauerhafter Funktionen zu verwalten und zu überwachen.

Ausführungen auflisten

Listet alle Ausführungen für eine dauerhafte Funktion auf:

```
aws lambda list-durable-executions-by-function \
  --function-name myDurableFunction
```

Details zur Ausführung abrufen

Rufen Sie Details zu einer bestimmten Ausführung ab:

```
aws lambda get-durable-execution \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

Ausführungshistorie abrufen

Den Checkpoint-Verlauf für eine Ausführung anzeigen:

```
aws lambda get-durable-execution-history \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

Stoppen Sie eine Ausführung

Stoppen Sie eine laufende dauerhafte Ausführung:

```
aws lambda stop-durable-execution \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

Funktionscode aktualisieren

Aktualisieren Sie Ihren dauerhaften Funktionscode und veröffentlichen Sie eine neue Version:

Um eine neue Version zu aktualisieren und zu veröffentlichen

1. Aktualisieren Sie den Funktionscode:

   ```
   aws lambda update-function-code \
     --function-name myDurableFunction \
     --zip-file fileb://function.zip
   ```

2. Warten Sie, bis das Update abgeschlossen ist:

   ```
   aws lambda wait function-updated \
     --function-name myDurableFunction
   ```

3. Veröffentlichen Sie eine neue Version:

   ```
   aws lambda publish-version \
     --function-name myDurableFunction \
     --description "Updated order processing logic"
   ```

4. Aktualisieren Sie den Alias so, dass er auf die neue Version verweist:

   ```
   aws lambda update-alias \
     --function-name myDurableFunction \
     --name prod \
   ```

```
  --function-version 2
```

> **Important**
>
> Laufende Ausführungen verwenden weiterhin die Version, mit der sie begonnen haben. Neue Aufrufe verwenden die aktualisierte Alias-Version.

Funktionsprotokolle anzeigen

Sehen Sie sich die Logs Ihrer dauerhaften Funktion unter Logs an CloudWatch :

```
aws logs tail /aws/lambda/myDurableFunction --follow
```

Filtern Sie die Protokolle nach einer bestimmten Ausführung:

```
aws logs filter-log-events \
  --log-group-name /aws/lambda/myDurableFunction \
  --filter-pattern "exec-abc123"
```

Bereinigen von Ressourcen

Löschen Sie Ihre dauerhafte Funktion und die zugehörigen Ressourcen:

```
# Delete the function
aws lambda delete-function --function-name myDurableFunction

# Delete the IAM role policies
aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy

# Delete the role
aws iam delete-role --role-name durable-function-role
```

Nächste Schritte

Nach der Bereitstellung Ihrer dauerhaften Funktion mit dem AWS CLI:

- Überwachen Sie Ausführungen mit den Befehlen `list-durable-executions-by-function` und `get-durable-execution`
- Checkpoint-Operationen in AWS CloudTrail Datenereignissen anzeigen
- Richten Sie CloudWatch Alarme für Ausführungsfehler oder lang andauernde Ausführungen ein
- Automatisieren Sie Bereitstellungen mithilfe von Shell-Skripten oder Pipelines CI/CD

Weitere Informationen zu AWS CLI Befehlen für Lambda finden Sie in der AWS CLI Befehlsreferenz.

Stellen Sie langlebige Lambda-Funktionen mit Infrastructure as Code bereit

Sie können langlebige Lambda-Funktionen mithilfe von Infrastructure as Code (IaC) -Tools wie AWS CloudFormation, AWS CDK AWS Serverless Application Model, oder Terraform bereitstellen. Mit diesen Tools können Sie Ihre Funktion, Ihre Ausführungsrolle und Ihre Berechtigungen im Code definieren, sodass Bereitstellungen wiederholbar und versionskontrolliert sind.

Für alle drei Tools müssen Sie:

- Ermöglichen Sie die dauerhafte Ausführung der Funktion
- Gewähren Sie der Ausführungsrolle Checkpoint-Berechtigungen
- Veröffentlichen Sie eine Version oder erstellen Sie einen Alias (dauerhafte Funktionen erfordern qualifizierte ARNs)

Dauerhafte Funktionen aus einer ZIP-Datei

AWS CloudFormation

Wird verwendet CloudFormation , um Ihre dauerhafte Funktion in einer Vorlage zu definieren. Das folgende Beispiel erstellt eine dauerhafte Funktion mit den erforderlichen Berechtigungen.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda durable function example

Resources:
  DurableFunctionRole:
    Type: AWS::IAM::Role
```

```
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy

  DurableFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      Role: !GetAtt DurableFunctionRole.Arn
      Code:
        ZipFile: |
          // Your durable function code here
          export const handler = async (event, context) => {
            return { statusCode: 200 };
          };
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7

  DurableFunctionVersion:
    Type: AWS::Lambda::Version
    Properties:
      FunctionName: !Ref DurableFunction
      Description: Initial version

  DurableFunctionAlias:
    Type: AWS::Lambda::Alias
    Properties:
      FunctionName: !Ref DurableFunction
      FunctionVersion: !GetAtt DurableFunctionVersion.Version
      Name: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
```

```
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunctionAlias
```

Um die Vorlage bereitzustellen

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name my-durable-function-stack \
  --capabilities CAPABILITY_IAM
```

AWS CDK

Mit AWS CDK können Sie die Infrastruktur mithilfe von Programmiersprachen definieren. Die folgenden Beispiele zeigen, wie eine dauerhafte Funktion mit TypeScript Python erstellt wird.

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.Function(this, 'DurableFunction', {
      runtime: lambda.Runtime.NODEJS_22_X,
      handler: 'index.handler',
      code: lambda.Code.fromAsset('lambda'),
      functionName: 'myDurableFunction',
      durableConfig: { executionTimeout: Duration.hours(1), retentionPeriod:
 Duration.days(30) },
    });

    // Create version and alias
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
```

```
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

Python

```
from aws_cdk import (
    Stack,
    aws_lambda as lambda_,
    aws_iam as iam,
    CfnOutput,
)
from constructs import Construct

class DurableFunctionStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        # Create the durable function
        durable_function = lambda_.Function(
            self, 'DurableFunction',
            runtime=lambda_.Runtime.NODEJS_22_X,
            handler='index.handler',
            code=lambda_.Code.from_asset('lambda'),
            function_name='myDurableFunction',
            durable_execution={execution_timeout: Duration.hours(1),
 retention_period: Duration.days(30)}
        )

        # Add durable execution managed policy for checkpoint permissions
        durable_function.role.add_managed_policy(
            iam.ManagedPolicy.from_aws_managed_policy_name('service-role/
AWSLambdaBasicDurableExecutionRolePolicy')
        )

        # Create version and alias
        version = durable_function.current_version
```

```
        alias = lambda_.Alias(
            self, 'ProdAlias',
            alias_name='prod',
            version=version
        )

        # Output the alias ARN
        CfnOutput(
            self, 'FunctionAliasArn',
            value=alias.function_arn,
            description='Use this ARN to invoke the durable function'
        )
```

Um den CDK-Stack bereitzustellen

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM vereinfacht CloudFormation Vorlagen für serverlose Anwendungen. Die folgende Vorlage erstellt eine dauerhafte Funktion mit AWS SAM.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      CodeUri: ./src
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7
      Policies:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy
      AutoPublishAlias: prod

Outputs:
```

```
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

Um die SAM-Vorlage bereitzustellen

```
sam build
sam deploy --guided
```

Terraform

Terraform ist ein beliebtes Open-Source-IaC-Tool, das Ressourcen unterstützt. AWS Das folgende Beispiel erstellt eine dauerhafte Funktion mit Terraform unter Verwendung der Provider-Version 6.25.0 oder höher. AWS

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.25.0"
    }
  }
}

provider "aws" {
  region = "us-east-2"
}

# IAM Role for Lambda Function
resource "aws_iam_role" "lambda_role" {
  name = "durable-function-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "lambda.amazonaws.com"
```

```
      }
    }]
  })
}

# Attach durable execution policy for checkpoint operations
resource "aws_iam_role_policy_attachment" "lambda_durable" {
  policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy"
  role       = aws_iam_role.lambda_role.name
}

# Lambda Function with Durable Execution enabled
resource "aws_lambda_function" "durable_function" {
  filename      = "function.zip"
  function_name = "myDurableFunction"
  role          = aws_iam_role.lambda_role.arn
  handler       = "index.handler"
  runtime       = "nodejs22.x"
  timeout       = 30
  memory_size   = 512

  durable_config {
    execution_timeout = 900
    retention_period  = 7
  }
}

# Publish a version
resource "aws_lambda_alias" "prod" {
  name             = "prod"
  function_name    = aws_lambda_function.durable_function.function_name
  function_version = aws_lambda_function.durable_function.version
}

output "function_arn" {
  description = "ARN of the Lambda function"
  value       = aws_lambda_function.durable_function.arn
}

output "alias_arn" {
  description = "ARN of the function alias (use this for invocations)"
  value       = aws_lambda_alias.prod.arn
```

```
}
```

Zur Bereitstellung mit Terraform

```
terraform init
terraform plan
terraform apply
```

> **Note**
>
> Für die Terraform-Unterstützung für langlebige Lambda-Funktionen ist die AWS Provider-Version 6.25.0 oder höher erforderlich. Aktualisieren Sie Ihre Anbieterversion, wenn Sie eine ältere Version verwenden.

Dauerhafte Funktionen aus einem OCI-Container-Image

Sie können Durable-Funktionen auch auf der Grundlage von Container-Images erstellen. Anweisungen zum Erstellen eines Container-Images finden Sie unter Unterstützte Laufzeiten für langlebige Funktionen.

AWS CDK

Mit AWS CDK können Sie die Infrastruktur mithilfe von Programmiersprachen definieren. Die folgenden Beispiele zeigen, wie Sie mithilfe TypeScript eines Container-Images eine dauerhafte Funktion erstellen.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.DockerImageFunction(this, 'DurableFunction', {
      code: lambda.DockerImageCode.fromImageAsset('./lambda', {
        platform: cdk.aws_ecr_assets.Platform.LINUX_AMD64,
      }),
```

```
      functionName: 'myDurableFunction',
      memorySize: 512,
      timeout: cdk.Duration.seconds(30),
      durableConfig: { executionTimeout: cdk.Duration.hours(1), retentionPeriod:
 cdk.Duration.days(30) },
    });

    // Create version and alias
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

Um den CDK-Stack bereitzustellen

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM vereinfacht CloudFormation Vorlagen für serverlose Anwendungen. Die folgende Vorlage erstellt eine dauerhafte Funktion mit AWS SAM.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      PackageType: Image
      ImageUri: ./src
      DurableConfig:
```

```
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7
      Policies:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy
      AutoPublishAlias: prod
    Metadata:
      DockerTag: latest
      DockerContext: ./src
      Dockerfile: Dockerfile

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

Um die SAM-Vorlage bereitzustellen

```
sam build
sam deploy --guided
```

Gängige Konfigurationsmuster

Unabhängig davon, welches IaC-Tool Sie verwenden, folgen Sie diesen Mustern, um dauerhafte Funktionen zu gewährleisten:

Ermöglichen Sie eine dauerhafte Ausführung

Stellen Sie die `DurableConfig` Eigenschaft für Ihre Funktion ein, um eine dauerhafte Ausführung zu ermöglichen. Diese Eigenschaft ist nur beim Erstellen der Funktion verfügbar. Sie können die dauerhafte Ausführung vorhandener Funktionen nicht aktivieren.

Gewähren Sie Checkpoint-Berechtigungen

Hängen Sie die `AWSLambdaBasicDurableExecutionRolePolicy` verwaltete Richtlinie an die Ausführungsrolle an. Diese Richtlinie umfasst die erforderlichen `lambda:CheckpointDurableExecution` und `lambda:GetDurableExecutionState` Berechtigungen.

Verwenden Sie qualifizierte ARNs

Erstellen Sie eine Version oder einen Alias für Ihre Funktion. Dauerhafte Funktionen benötigen qualifizierte ARNs (mit Version oder Alias) für den Aufruf. Verwenden Sie `AutoPublishAlias` in, und Terraform AWS SAM oder erstellen Sie explizite Versionen. CloudFormation AWS CDK

Abhängigkeiten von Paketen

Nehmen Sie das Durable Execution SDK in Ihr Bereitstellungspaket auf. Für Node.js, installieren`@aws/durable-execution-sdk-js`. Für Python installieren Sie`aws-durable-execution-sdk-python`.

Nächste Schritte

Nach der Bereitstellung Ihrer dauerhaften Funktion:

- Testen Sie Ihre Funktion mit dem qualifizierten ARN (Version oder Alias)
- Überwachen Sie den Ausführungsfortschritt in der Lambda-Konsole auf der Registerkarte Dauerhafte Ausführungen
- Zeigen Sie Checkpoint-Operationen in Datenereignissen an AWS CloudTrail
- Prüfen Sie die CloudWatch Protokolle auf Funktionsausgabe und Wiedergabeverhalten

Weitere Informationen zur Bereitstellung von Lambda-Funktionen mit IaC-Tools finden Sie unter:

- CloudFormation AWS::Lambda: :Funktionsreferenz
- AWS CDK Dokumentation zum Lambda-Modul
- AWS SAM Entwicklerhandbuch

Dauerhafte Lambda-Funktionen konfigurieren

Die Einstellungen für die dauerhafte Ausführung steuern, wie lange eine dauerhafte Ausführung über alle Aufrufe hinweg ausgeführt werden kann, wie lange der Ausführungsverlauf AWS Lambda beibehalten wird und welcher AWS KMS Schlüssel Ihre Ausführungsdaten verschlüsselt. Konfigurieren Sie diese Einstellungen, um eine dauerhafte Ausführung für Ihre Funktion zu ermöglichen.

Dauerhafte Ausführung aktivieren

Konfigurieren Sie das `DurableConfig` Objekt bei der Erstellung Ihrer Funktion so, dass es ein Timeout für die Ausführung und die Aufbewahrung des Verlaufs festlegt. Sie können die dauerhafte

Ausführung nur aktivieren, wenn Sie eine Funktion erstellen. Sie können es nicht für bestehende Funktionen aktivieren.

AWS CLI

```
aws lambda create-function \
  --function-name my-durable-function \
  --runtime nodejs24.x \
  --role arn:aws:iam::123456789012:role/my-durable-role \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 30,
 "KMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id"}'
```

CloudFormation

```
Resources:
  MyDurableFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-durable-function
      Runtime: nodejs24.x
      Handler: index.handler
      Code:
        ZipFile: |
          // Your durable function code
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 30
        KMSKeyArn: "arn:aws:kms:us-east-1:111122223333:key/key-id"
```

Konfigurationsparameter:

- `ExecutionTimeout` – Erforderlich. Die maximale Zeit in Sekunden, die eine dauerhafte Ausführung ausgeführt werden kann, bevor Lambda die Ausführung stoppt. Dieses Timeout gilt für die gesamte dauerhafte Ausführung, nicht für einzelne Funktionsaufrufe. Sie müssen angeben, `ExecutionTimeout` wann immer Sie es angeben. Das Erstellen einer Funktion mit

einer dauerhaften Konfiguration`DurableConfig`, aber ohne Ausführungs-Timeout schlägt mit einem fehl. `InvalidParameterValueException` Gültiger Bereich: 1—31622400.

- `RetentionPeriodInDays` – Optional. Die Anzahl der Tage, für die der Ausführungsverlauf nach Abschluss einer dauerhaften Ausführung beibehalten werden soll. Nach diesem Zeitraum ist der Ausführungsverlauf nicht mehr über die `GetDurableExecutionHistory` API verfügbar. Gültiger Bereich: 1—90. Standard: 14.
- `KMSKeyArn` – Optional. Der ARN eines vom Kunden verwalteten Schlüssels zur Verschlüsselung dauerhafter Ausführungsdaten. Weitere Informationen finden Sie unter Verschlüsseln von dauerhaften Lambda-Ausführungsdaten.

Die vollständige API-Referenz finden Sie DurableConfigin der Lambda-API-Referenz.

Bewährte Methoden für die Konfiguration

Folgen Sie diesen bewährten Methoden, wenn Sie langlebige Funktionen für den Produktionseinsatz konfigurieren:

- Legen Sie geeignete Zeitlimits für die Ausführung fest — richten Sie die Konfiguration auf der `ExecutionTimeout` Grundlage der maximal zu erwartenden Dauer Ihres Workflows aus. Legen Sie keine unnötig langen Timeouts fest, da sich diese auf die Kosten und die Ressourcenzuweisung auswirken.
- Sorgen Sie für ein ausgewogenes Verhältnis zwischen Aufbewahrung und Speicherkosten — Die Festlegung erfolgt auf der `RetentionPeriodInDays` Grundlage Ihrer Debugging- und Audit-Anforderungen. Längere Aufbewahrungsfristen erhöhen die Speicherkosten.
- Statusgröße überwachen — Große Zustandsobjekte erhöhen die Speicherkosten und können die Leistung beeinträchtigen. Halten Sie den Status minimal und verwenden Sie externen Speicher für große Datenmengen.
- Konfigurieren Sie die entsprechende Protokollierung — Aktivieren Sie die detaillierte Protokollierung zur Fehlerbehebung bei Workflows mit langer Laufzeit. Berücksichtigen Sie dabei jedoch die Auswirkungen auf das Protokollvolumen und die Kosten.
- Mit Ihrem eigenen AWS KMS Schlüssel verschlüsseln — Wählen Sie einen vom `KMSKeyArn` Kunden verwalteten Schlüssel, um die Schlüsselrotation zu kontrollieren, den Zugriff zu überprüfen und Compliance-Anforderungen zu erfüllen. Siehe Verschlüsseln AWS Lambda dauerhafte Ausführungsdaten.

Beispiel für eine Produktionskonfiguration:

```
{
  "ExecutionTimeout": 86400,
  "RetentionPeriodInDays": 7,
  "KMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id"
}
```

In diesem Beispiel wird ein Ausführungstimeout von 24 Stunden (86.400 Sekunden), eine Aufbewahrungsfrist von 7 Tagen und ein vom Kunden verwalteter Schlüssel für die Verschlüsselung festgelegt.

Dauerhafte Funktionen oder Step Functions

Beide Funktionen von Lambda sind dauerhaft und AWS Step Functions ermöglichen eine zuverlässige Workflow-Orchestrierung mit automatischem Statusmanagement und Wiederherstellung nach einem Ausfall. Sie dienen unterschiedlichen Entwicklerpräferenzen und Architekturmustern. Dauerhafte Funktionen sind für die Anwendungsentwicklung in Lambda optimiert, während Step Functions für die dienstübergreifende AWS Workflow-Orchestrierung konzipiert ist.

Wann sollten langlebige Funktionen verwendet werden

Verwenden Sie dauerhafte Funktionen, wenn:

- Ihr Team bevorzugt Standardprogrammiersprachen und vertraute Entwicklungstools
- Ihre Anwendungslogik befindet sich hauptsächlich in Lambda-Funktionen
- Sie möchten eine genaue Kontrolle über den Ausführungsstatus im Code
- Sie entwickeln Lambda-zentrierte Anwendungen mit enger Kopplung zwischen Workflow und Geschäftslogik
- Sie möchten schnell iterieren, ohne zwischen Code und Designern hin- und herwechseln zu müssen visual/JSON

Wann sollte Step Functions verwendet werden?

Verwenden Sie Step Functions, wenn:

- Sie benötigen eine visuelle Workflow-Darstellung für teamübergreifende Sichtbarkeit

- Sie orchestrieren mehrere AWS Dienste und wünschen sich native Integrationen ohne benutzerdefinierten SDK-Code
- Sie benötigen eine wartungsfreie Infrastruktur (kein Patchen, Runtime-Updates)
- Nicht technisch versierte Akteure müssen die Workflow-Logik verstehen und validieren

Entscheidungsrahmen

Ermitteln Sie anhand der folgenden Fragen, welcher Service zu Ihrem Anwendungsfall passt:

- Was ist Ihr Hauptaugenmerk? Anwendungsentwicklung in Lambda → langlebige Funktionen. Workflow-Orchestrierung über AWS → Step Functions hinweg.
- Was ist Ihr bevorzugtes Programmiermodell? Standard-Programmiersprachen → langlebige Funktionen. Graphbasiertes DSL oder visueller Designer → Step Functions.
- Wie viele AWS Dienste sind betroffen? In erster Linie Lambda → langlebige Funktionen. Mehrere AWS Dienste → Step Functions.
- Welche Entwicklungstools verwenden Sie? Erfahrung als Lambda-Entwickler, IDE mit LLM-Agent, programmiersprachenspezifische Unit-Test-Frameworks,, AWS SAM AWS CDK, AWS Toolkit → langlebige Funktionen. Visual Workflow Builder, AWS CDK um Workflows zu modellieren → Step Functions.
- Wer verwaltet die Infrastruktur? Sie wünschen sich Flexibilität innerhalb von Lambda → langlebige Funktionen. Sie möchten vollständig verwaltete, wartungsfreie → Step Functions.

Funktionsvergleich

In der folgenden Tabelle werden die wichtigsten Funktionen von Step Functions und Lambda Durable Functions verglichen:

Feature	AWS Step Functions	Langlebige Lambda-Funktionen
Primärer Fokus	Workflow-Orchestrierung auf allen Ebenen AWS	Anwendungsentwicklung in Lambda
Servicetyp	Eigenständiger, dedizierter Workflow-Service	Läuft innerhalb von Lambda

Feature	AWS Step Functions	Langlebige Lambda-Funktionen
Programmiermodell	Graphbasiert, DSL in den Amazon-Staaten oder AWS CDK	Standard-Programmiersprachen (JavaScript/TypeScript, Python)
Entwicklungstools	Visual Builder in der Konsole// AWS Toolkit-IDE-Erweiterung, AWS CDK	Lambda DX innerhalb von IDE- und LLM-Agenten, Unit-Test-Frameworks AWS SAM, AWS Toolkit-IDE-Erweiterung
Integrationen	Über 220 AWS Dienste, 16.000 APIs	Modellerweiterung für ereignisg esteuerte Lambda-Programmier ung (Ereignisquellen)
Verwaltung	Vollständig verwaltet, laufzeitu nabhängig, wartungsfrei (kein Patchen, Runtime-Updates)	In der Lambda-Umgebung verwaltet
Am besten geeignet für	Geschäftsprozess- und IT-Automa tisierung, Datenverarbeitung, KI-Workflows	Verteilte Transaktionen, zustandso rientierte Anwendungslogik, Funktionsorchestrierung, Datenverarbeitung, KI-Workflows

Hybride Architekturen

Viele Anwendungen profitieren von der Nutzung beider Dienste. Ein gängiges Muster ist die Verwendung dauerhafter Funktionen für die Logik auf Anwendungsebene innerhalb von Lambda, während Step Functions Workflows auf hoher Ebene über mehrere AWS Dienste hinweg koordiniert, die über Lambda-Funktionen hinausgehen.

Überlegungen zur Migration

Einfach anfangen, komplex werden: Beginnen Sie mit dauerhaften Funktionen für Lambda-zentrierte Workflows. Fügen Sie Step Functions hinzu, wenn Sie Multiservice-Orchestrierung oder visuelles Workflow-Design benötigen.

Bestehende Step Functions-Benutzer: Keep Step Functions für etablierte dienstübergreifende Workflows. Ziehen Sie langlebige Funktionen für eine neue Lambda-Anwendungslogik in Betracht, die Zuverlässigkeit erfordert.

Zugehörige Ressourcen

- Langlebige Lambda-Funktionen
- Orchestrierung von Lambda-Funktionen mit Step Functions
- Erste Schritte mit dauerhaften Funktionen

Beispiele und Anwendungsfälle

Mit den dauerhaften Lambda-Funktionen können Sie fehlertolerante, mehrstufige Anwendungen mit dauerhaften Vorgängen wie Steps und Waits erstellen. Mit automatischem Checkpointing und einem Checkpoint-Replay-Modell, bei dem die Ausführung nach einem Ausfall wieder von vorne beginnt, abgeschlossene Checkpoints jedoch überspringt, können Ihre Funktionen nach Ausfällen wiederhergestellt und die Ausführung wieder aufgenommen werden, ohne dass der Fortschritt verloren geht.

> **Note**
>
> Die SDK-API-Referenz, Schnellstart-Tutorials, sprachspezifische Anleitungen und zusätzliche Workflow-Muster finden Sie im Durable Execution SDK Developer Guide.AWS

Short-lived fehlertolerante Prozesse

Verwenden Sie langlebige Funktionen, um zuverlässige Abläufe aufzubauen, die in der Regel innerhalb von Minuten abgeschlossen werden. Diese Prozesse sind zwar kürzer als langwierige Workflows, profitieren aber dennoch von automatischem Checkpoint und Fehlertoleranz in verteilten Systemen. Zuverlässige Funktionen tragen dazu bei, dass Ihre mehrstufigen Prozesse auch dann erfolgreich abgeschlossen werden, wenn einzelne Serviceanrufe fehlschlagen, ohne dass komplexe Fehlerbehandlungs- oder Statusverwaltungscodes erforderlich sind.

Zu den gängigen Szenarien gehören Hotelbuchungssysteme, Restaurantreservierungsplattformen, Anfragen für Mitfahrgelegenheiten, der Kauf von Veranstaltungstickets und SaaS-Abonnement-Upgrades. Diese Szenarien weisen gemeinsame Merkmale auf: mehrere Serviceanrufe, die zusammen abgeschlossen werden müssen, die Notwendigkeit, bei vorübergehenden Ausfällen automatische Wiederholungsversuche durchzuführen, und die Anforderung, einen konsistenten Status über verteilte Systeme hinweg aufrechtzuerhalten.

Verteilte Transaktionen über Microservices

Koordinieren Sie Zahlungen, Inventar und Versand für mehrere Dienste mit automatischem Rollback bei Ausfällen. Jeder Servicevorgang ist in einem Schritt zusammengefasst, sodass sichergestellt wird, dass die Transaktion jederzeit wiederhergestellt werden kann, falls ein Service ausfällt.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, amount, items } = event;

    // Reserve inventory across multiple warehouses
    const inventory = await context.step("reserve-inventory", async () => {
      return await inventoryService.reserve(items);
    });

    // Process payment
    const payment = await context.step("process-payment", async () => {
      return await paymentService.charge(amount);
    });

    // Create shipment
    const shipment = await context.step("create-shipment", async () => {
      return await shippingService.createShipment(orderId, inventory);
    });

    return { orderId, status: 'completed', shipment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
```

```
    order_id = event['orderId']
    amount = event['amount']
    items = event['items']

    # Reserve inventory across multiple warehouses
    inventory = context.step(
        lambda _: inventory_service.reserve(items),
        name='reserve-inventory'
    )

    # Process payment
    payment = context.step(
        lambda _: payment_service.charge(amount),
        name='process-payment'
    )

    # Create shipment
    shipment = context.step(
        lambda _: shipping_service.create_shipment(order_id, inventory),
        name='create-shipment'
    )

    return {'orderId': order_id, 'status': 'completed', 'shipment': shipment}
```

Schlägt ein Schritt fehl, wiederholt die Funktion den Vorgang automatisch vom letzten erfolgreichen Checkpoint aus. Die Inventarreservierung bleibt auch dann bestehen, wenn die Zahlungsabwicklung vorübergehend fehlschlägt. Wenn die Funktion es erneut versucht, überspringt sie den abgeschlossenen Inventarisierungsschritt und fährt direkt mit der Zahlungsabwicklung fort. Dadurch werden doppelte Reservierungen vermieden und ein einheitlicher Status in Ihrem gesamten verteilten System gewährleistet.

Auftragsabwicklung mit mehreren Schritten

Bearbeiten Sie Bestellungen durch Validierung, Zahlungsautorisierung, Inventarzuweisung und Versand mit automatischer Wiederholung und Wiederherstellung. Jeder Schritt wird überprüft, sodass sichergestellt wird, dass die Bestellung auch dann bearbeitet wird, wenn einzelne Schritte fehlschlagen und es erneut versucht wird.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId, items } = event;

    // Validate order details
    const validation = await context.step("validate-order", async () => {
      const customer = await customerService.validate(customerId);
      const itemsValid = await inventoryService.validateItems(items);
      return { customer, itemsValid };
    });

    if (!validation.itemsValid) {
      return { orderId, status: 'rejected', reason: 'invalid_items' };
    }

    // Authorize payment
    const authorization = await context.step("authorize-payment", async () => {
      return await paymentService.authorize(
        validation.customer.paymentMethod,
        calculateTotal(items)
      );
    });

    // Allocate inventory
    const allocation = await context.step("allocate-inventory", async () => {
      return await inventoryService.allocate(items);
    });

    // Fulfill order
    const fulfillment = await context.step("fulfill-order", async () => {
      return await fulfillmentService.createShipment({
        orderId,
        items: allocation.allocatedItems,
        address: validation.customer.shippingAddress
      });
    });

    return {
```

```
      orderId,
      status: 'completed',
      trackingNumber: fulfillment.trackingNumber
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']
    items = event['items']

    # Validate order details
    def validate_order(_):
        customer = customer_service.validate(customer_id)
        items_valid = inventory_service.validate_items(items)
        return {'customer': customer, 'itemsValid': items_valid}

    validation = context.step(validate_order, name='validate-order')

    if not validation['itemsValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'invalid_items'}

    # Authorize payment
    authorization = context.step(
        lambda _: payment_service.authorize(
            validation['customer']['paymentMethod'],
            calculate_total(items)
        ),
        name='authorize-payment'
    )

    # Allocate inventory
    allocation = context.step(
        lambda _: inventory_service.allocate(items),
```

```
        name='allocate-inventory'
    )

    # Fulfill order
    fulfillment = context.step(
        lambda _: fulfillment_service.create_shipment({
            'orderId': order_id,
            'items': allocation['allocatedItems'],
            'address': validation['customer']['shippingAddress']
        }),
        name='fulfill-order'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': fulfillment['trackingNumber']
    }
```

Dieses Muster stellt sicher, dass Bestellungen niemals in Zwischenzuständen stecken bleiben. Schlägt die Validierung fehl, wird die Bestellung vor der Zahlungsautorisierung abgelehnt. Wenn die Zahlungsautorisierung fehlschlägt, wird kein Inventar zugewiesen. Jeder Schritt baut auf dem vorherigen auf und bietet automatische Wiederholungen und Wiederherstellungen.

Hinweis

Die bedingte Prüfung `if (!validation.itemsValid)` erfolgt außerhalb eines Schritts und wird während der Wiedergabe erneut ausgeführt. Das ist sicher, weil es deterministisch ist — es liefert immer dasselbe Ergebnis bei demselben Überprüfungsobjekt.

Long-running Prozesse

Verwenden Sie langlebige Funktionen für Prozesse, die sich über Stunden, Tage oder Wochen erstrecken. Wartevorgänge unterbrechen die Ausführung, ohne dass Rechenkosten anfallen, wodurch Prozesse mit langer Laufzeit kostengünstig werden. Während Wartezeiten wird Ihre Funktion nicht mehr ausgeführt und Lambda recycelt die Ausführungsumgebung. Wenn es Zeit ist, fortzufahren, ruft Lambda Ihre Funktion erneut auf und spielt sie ab dem letzten Checkpoint erneut ab.

Dieses Ausführungsmodell macht langlebige Funktionen ideal für Prozesse, die über einen längeren Zeitraum unterbrochen werden müssen, unabhängig davon, ob sie auf menschliche Entscheidungen, externe Systemreaktionen, geplante Verarbeitungsfenster oder zeitbedingte Verzögerungen warten. Sie zahlen nur für aktive Rechenzeit, nicht für Wartezeiten.

Zu den gängigen Szenarien gehören Prozesse zur Genehmigung von Dokumenten, geplante Stapelverarbeitung, mehrtägige Onboarding-Prozesse, Abonnement-Testprozesse und Systeme für verzögerte Benachrichtigungen. Diese Szenarien haben gemeinsame Merkmale: längere Wartezeiten, gemessen in Stunden oder Tagen, die Notwendigkeit, den Ausführungsstatus während dieser Wartezeiten beizubehalten, und kostensensible Anforderungen, bei denen es unerschwinglich ist, für ungenutzte Rechenzeit zu bezahlen.

Human-in-the-loop Zulassungen

Unterbrechen Sie die Ausführung für Dokumentenprüfungen, Genehmigungen oder Entscheidungen und behalten Sie dabei den Ausführungsstatus bei. Die Funktion wartet auf externe Rückrufe, ohne Ressourcen zu verbrauchen, und wird automatisch fortgesetzt, wenn die Genehmigung eingeht.

Dieses Muster ist wichtig für Prozesse, die menschliches Urteilsvermögen oder externe Validierung erfordern. Die Funktion wird am Rückrufpunkt unterbrochen, sodass während des Wartens keine Rechenkosten anfallen. Wenn jemand seine Entscheidung über die API übermittelt, ruft Lambda Ihre Funktion erneut auf und spielt sie vom Checkpoint aus erneut ab, wobei das Genehmigungsergebnis angezeigt wird.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { documentId, reviewers } = event;

    // Step 1: Prepare document for review
    const prepared = await context.step("prepare-document", async () => {
      return await documentService.prepare(documentId);
    });

    // Step 2: Request approval with callback
    const approval = await context.waitForCallback(
```

```
      "approval-callback",
      async (callbackId) => {
        await notificationService.sendApprovalRequest({
          documentId,
          reviewers,
          callbackId,
          expiresIn: 86400
        });
      },
      {
        timeout: { seconds: 86400 }
      }
    );

    // Function resumes here when approval is received
    if (approval?.approved) {
      const finalized = await context.step("finalize-document", async () => {
        return await documentService.finalize(documentId, approval.comments);
      });

      return {
        status: 'approved',
        documentId,
        finalizedAt: finalized.timestamp
      };
    }

    // Handle rejection
    await context.step("archive-rejected", async () => {
      await documentService.archive(documentId, approval?.reason);
    });

    return {
      status: 'rejected',
      documentId,
      reason: approval?.reason
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 WaitConfig

@durable_execution
def lambda_handler(event, context: DurableContext):
    document_id = event['documentId']
    reviewers = event['reviewers']

    # Step 1: Prepare document for review
    prepared = context.step(
        lambda _: document_service.prepare(document_id),
        name='prepare-document'
    )

    # Step 2: Request approval with callback
    def send_approval_request(callback_id):
        notification_service.send_approval_request({
            'documentId': document_id,
            'reviewers': reviewers,
            'callbackId': callback_id,
            'expiresIn': 86400
        })

    approval = context.wait_for_callback(
        send_approval_request,
        name='approval-callback',
        config=WaitConfig(timeout=86400)
    )

    # Function resumes here when approval is received
    if approval and approval.get('approved'):
        finalized = context.step(
            lambda _: document_service.finalize(document_id,
 approval.get('comments')),
            name='finalize-document'
        )

        return {
            'status': 'approved',
            'documentId': document_id,
            'finalizedAt': finalized['timestamp']
```

```
        }

    # Handle rejection
    context.step(
        lambda _: document_service.archive(document_id, approval.get('reason') if
 approval else None),
        name='archive-rejected'
    )

    return {
        'status': 'rejected',
        'documentId': document_id,
        'reason': approval.get('reason') if approval else None
    }
```

Wenn der Rückruf eingeht und Ihre Funktion wieder aufgenommen wird, wird er von Anfang an wiedergegeben. Der Schritt „Dokument vorbereiten“ gibt sofort das Ergebnis der Prüfung zurück. Der ForCallback Wartevorgang kehrt ebenfalls sofort mit dem gespeicherten Genehmigungsergebnis zurück, anstatt erneut zu warten. Die Ausführung wird dann mit den Finalisierungs- oder Archivierungsschritten fortgesetzt.

Multi-stage Daten-Pipelines

Verarbeiten Sie große Datensätze durch Extraktions-, Transformations- und Ladephasen mit Checkpoints zwischen den Phasen. Jede Phase kann Stunden in Anspruch nehmen, und Checkpoints ermöglichen es, dass die Pipeline bei einer Unterbrechung aus jeder Phase wieder aufgenommen werden kann.

Dieses Muster ist ideal für ETL-Workflows, Datenmigrationen oder Batch-Verarbeitungsaufträge, bei denen Sie Daten stufenweise mit Wiederherstellungspunkten zwischen ihnen verarbeiten müssen. Wenn eine Phase fehlschlägt, wird die Pipeline mit der letzten abgeschlossenen Phase fortgesetzt, anstatt von vorne neu zu starten. Sie können Warteoperationen auch verwenden, um zwischen den Phasen eine Pause einzulegen. Dabei können Sie Ratenbegrenzungen einhalten, warten, bis nachgelagerte Systeme bereit sind, oder die Verarbeitung außerhalb der Spitzenzeiten planen.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize } = event;

    // Stage 1: Extract data from source
    const extracted = await context.step("extract-data", async () => {
      const records = await sourceDatabase.extractRecords(datasetId);
      return { recordCount: records.length, records };
    });

    // Wait 5 minutes to respect source system rate limits
    await context.wait({ seconds: 300 });

    // Stage 2: Transform data in batches
    const transformed = await context.step("transform-data", async () => {
      const batches = chunkArray(extracted.records, batchSize);
      const results = [];

      for (const batch of batches) {
        const transformed = await transformService.processBatch(batch);
        results.push(transformed);
      }

      return { batchCount: batches.length, results };
    });

    // Wait until off-peak hours (e.g., 2 AM)
    const now = new Date();
    const targetHour = 2;
    const msUntilTarget = calculateMsUntilHour(now, targetHour);
    await context.wait({ seconds: Math.floor(msUntilTarget / 1000) });

    // Stage 3: Load data to destination
    const loaded = await context.step("load-data", async () => {
      let loadedCount = 0;

      for (const result of transformed.results) {
        await destinationDatabase.loadBatch(result);
        loadedCount += result.length;
      }

      return { loadedCount };
    });
```

```
    // Stage 4: Verify and finalize
    const verified = await context.step("verify-pipeline", async () => {
      const verification = await destinationDatabase.verifyRecords(datasetId);
      await pipelineService.markComplete(datasetId, verification);
      return verification;
    });

    return {
      datasetId,
      recordsProcessed: extracted.recordCount,
      batchesProcessed: transformed.batchCount,
      recordsLoaded: loaded.loadedCount,
      verified: verified.success
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event['batchSize']

    # Stage 1: Extract data from source
    def extract_data(_):
        records = source_database.extract_records(dataset_id)
        return {'recordCount': len(records), 'records': records}

    extracted = context.step(extract_data, name='extract-data')

    # Wait 5 minutes to respect source system rate limits
    context.wait(Duration.from_seconds(300))

    # Stage 2: Transform data in batches
    def transform_data(_):
        batches = chunk_array(extracted['records'], batch_size)
```

```
        results = []

        for batch in batches:
            transformed = transform_service.process_batch(batch)
            results.append(transformed)

        return {'batchCount': len(batches), 'results': results}

    transformed = context.step(transform_data, name='transform-data')

    # Wait until off-peak hours (e.g., 2 AM)
    now = datetime.now()
    target_hour = 2
    ms_until_target = calculate_ms_until_hour(now, target_hour)
    context.wait(ms_until_target // 1000)

    # Stage 3: Load data to destination
    def load_data(_):
        loaded_count = 0

        for result in transformed['results']:
            destination_database.load_batch(result)
            loaded_count += len(result)

        return {'loadedCount': loaded_count}

    loaded = context.step(load_data, name='load-data')

    # Stage 4: Verify and finalize
    def verify_pipeline(_):
        verification = destination_database.verify_records(dataset_id)
        pipeline_service.mark_complete(dataset_id, verification)
        return verification

    verified = context.step(verify_pipeline, name='verify-pipeline')

    return {
        'datasetId': dataset_id,
        'recordsProcessed': extracted['recordCount'],
        'batchesProcessed': transformed['batchCount'],
        'recordsLoaded': loaded['loadedCount'],
        'verified': verified['success']
    }
```

Jede Phase ist in einem Schritt zusammengefasst, wodurch ein Checkpoint entsteht, über den die Pipeline bei einer Unterbrechung aus jeder Phase wieder aufgenommen werden kann. Bei der 5-minütigen Wartezeit zwischen Extrahieren und Transformieren werden die Geschwindigkeitsbegrenzungen des Quellsystems eingehalten, ohne Rechenressourcen zu verbrauchen, während bei der Wartezeit bis 2 Uhr der teure Ladevorgang außerhalb der Spitzenzeiten geplant wird.

Hinweis

Der `new Date()` Aufruf und die `calculateMsUntilHour()` Funktion befinden sich außerhalb von Schritten und werden während der Wiedergabe erneut ausgeführt. Für zeitbasierte Operationen, die bei allen Wiederholungen konsistent sein müssen, berechnen Sie den Zeitstempel innerhalb eines Schritts oder verwenden Sie ihn nur für Wartezeiten (die mit Checkpoints versehen sind).

Fortgeschrittene Muster

Verwenden Sie langlebige Funktionen, um komplexe mehrstufige Anwendungen zu erstellen, die mehrere dauerhafte Operationen, parallel Ausführung, Array-Verarbeitung, bedingte Logik und Polling kombinieren. Mit diesen Mustern können Sie anspruchsvolle Anwendungen erstellen, die viele Aufgaben koordinieren und gleichzeitig die Fehlertoleranz und die automatische Wiederherstellung wahren.

Fortgeschrittene Muster gehen über einfache sequentielle Schritte hinaus. Sie können Operationen gleichzeitig mit diesen Primitiven ausführen`parallel()`, Arrays mit verarbeiten`map()`, auf externe Bedingungen warten und diese Primitive kombinieren`waitForCondition()`, um zuverlässige Anwendungen zu erstellen. Jeder dauerhafte Vorgang erstellt seine eigenen Prüfpunkte, sodass Ihre Anwendung bei einer Unterbrechung jederzeit wiederhergestellt werden kann.

Onboarding-Prozesse für Benutzer

Führen Sie Benutzer durch die Registrierung, E-Mail-Überprüfung, Profileinrichtung und Erstkonfiguration mit Wiederholungsversuchen. In diesem Beispiel werden aufeinanderfolgende Schritte, Rückrufe und bedingte Logik kombiniert, um einen vollständigen Onboarding-Prozess zu erstellen.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { userId, email } = event;

    // Step 1: Create user account
    const user = await context.step("create-account", async () => {
      return await userService.createAccount(userId, email);
    });

    // Step 2: Send verification email
    await context.step("send-verification", async () => {
      return await emailService.sendVerification(email);
    });

    // Step 3: Wait for email verification (up to 48 hours)
    const verified = await context.waitForCallback(
      "email-verification",
      async (callbackId) => {
        await notificationService.sendVerificationLink({
          email,
          callbackId,
          expiresIn: 172800
        });
      },
      {
        timeout: { seconds: 172800 }
      }
    );

    if (!verified) {
      await context.step("send-reminder", async () => {
        await emailService.sendReminder(email);
      });

      return {
        status: "verification_timeout",
        userId,
        message: "Email verification not completed within 48 hours"
      };
    }
```

```
    // Step 4: Initialize user profile in parallel
    const setupResults = await context.parallel("profile-setup", [
      async (ctx: DurableContext) => {
        return await ctx.step("create-preferences", async () => {
          return await preferencesService.createDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("setup-notifications", async () => {
          return await notificationService.setupDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("create-welcome-content", async () => {
          return await contentService.createWelcome(userId);
        });
      }
    ]);

    // Step 5: Send welcome email
    await context.step("send-welcome", async () => {
      const [preferences, notifications, content] = setupResults.getResults();
      return await emailService.sendWelcome({
        email,
        preferences,
        notifications,
        content
      });
    });

    return {
      status: "onboarding_complete",
      userId,
      completedAt: new Date().toISOString()
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 WaitConfig
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    user_id = event['userId']
    email = event['email']

    # Step 1: Create user account
    user = context.step(
        lambda _: user_service.create_account(user_id, email),
        name='create-account'
    )

    # Step 2: Send verification email
    context.step(
        lambda _: email_service.send_verification(email),
        name='send-verification'
    )

    # Step 3: Wait for email verification (up to 48 hours)
    def send_verification_link(callback_id):
        notification_service.send_verification_link({
            'email': email,
            'callbackId': callback_id,
            'expiresIn': 172800
        })

    verified = context.wait_for_callback(
        send_verification_link,
        name='email-verification',
        config=WaitConfig(timeout=172800)
    )

    if not verified:
        context.step(
            lambda _: email_service.send_reminder(email),
            name='send-reminder'
        )
```

```
        return {
            'status': 'verification_timeout',
            'userId': user_id,
            'message': 'Email verification not completed within 48 hours'
        }

    # Step 4: Initialize user profile in parallel
    def create_preferences(ctx: DurableContext):
        return ctx.step(
            lambda _: preferences_service.create_defaults(user_id),
            name='create-preferences'
        )

    def setup_notifications(ctx: DurableContext):
        return ctx.step(
            lambda _: notification_service.setup_defaults(user_id),
            name='setup-notifications'
        )

    def create_welcome_content(ctx: DurableContext):
        return ctx.step(
            lambda _: content_service.create_welcome(user_id),
            name='create-welcome-content'
        )

    setup_results = context.parallel(
        [create_preferences, setup_notifications, create_welcome_content],
        name='profile-setup'
    )

    # Step 5: Send welcome email
    def send_welcome(_):
        results = setup_results.get_results()
        preferences, notifications, content = results[0], results[1], results[2]
        return email_service.send_welcome({
            'email': email,
            'preferences': preferences,
            'notifications': notifications,
            'content': content
        })

    context.step(send_welcome, name='send-welcome')

    return {
```

```
        'status': 'onboarding_complete',
        'userId': user_id,
        'completedAt': datetime.now().isoformat()
    }
```

Der Prozess kombiniert aufeinanderfolgende Schritte mit Checkpoints für die Kontoerstellung und das Senden von E-Mails und wartet dann bis zu 48 Stunden lang auf die E-Mail-Bestätigung, ohne Ressourcen zu verbrauchen. Die bedingte Logik verarbeitet unterschiedliche Pfade, je nachdem, ob die Überprüfung abgeschlossen ist oder ob das Zeitlimit überschritten wird. Aufgaben zur Profileinrichtung werden gleichzeitig mit parallel Vorgängen ausgeführt, um die Gesamtausführungszeit zu reduzieren. Jeder Schritt wird bei vorübergehenden Ausfällen automatisch wiederholt, um sicherzustellen, dass das Onboarding zuverlässig abgeschlossen wird.

Funktionsübergreifende verkettete Aufrufe

Rufen Sie andere Lambda-Funktionen aus einer dauerhaften Funktion heraus auf, indem Sie. `context.invoke()` Die aufrufende Funktion wird angehalten, während auf den Abschluss der aufgerufenen Funktion gewartet wird, wodurch ein Checkpoint erstellt wird, der das Ergebnis beibehält. Wenn die aufrufende Funktion nach Abschluss der aufgerufenen Funktion unterbrochen wird, wird sie mit dem gespeicherten Ergebnis fortgesetzt, ohne die Funktion erneut aufzurufen.

Verwenden Sie dieses Muster, wenn Sie über spezielle Funktionen verfügen, die sich mit bestimmten Bereichen befassen (Kundenvalidierung, Zahlungsabwicklung, Bestandsverwaltung) und diese in einem Workflow koordinieren müssen. Jede Funktion behält ihre eigene Logik bei und kann von mehreren Orchestrator-Funktionen aufgerufen werden, wodurch Codeduplikationen vermieden werden.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

// Main orchestrator function
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId } = event;

    // Step 1: Validate customer by invoking customer service function
```

```
    const customer = await context.invoke(
      "validate-customer",
      "arn:aws:lambda:us-east-1:123456789012:function:customer-service:1",
      { customerId }
    );

    if (!customer.isValid) {
      return { orderId, status: "rejected", reason: "invalid_customer" };
    }

    // Step 2: Check inventory by invoking inventory service function
    const inventory = await context.invoke(
      "check-inventory",
      "arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1",
      { orderId, items: event.items }
    );

    if (!inventory.available) {
      return { orderId, status: "rejected", reason: "insufficient_inventory" };
    }

    // Step 3: Process payment by invoking payment service function
    const payment = await context.invoke(
      "process-payment",
      "arn:aws:lambda:us-east-1:123456789012:function:payment-service:1",
      {
        customerId,
        amount: inventory.totalAmount,
        paymentMethod: customer.paymentMethod
      }
    );

    // Step 4: Create shipment by invoking fulfillment service function
    const shipment = await context.invoke(
      "create-shipment",
      "arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1",
      {
        orderId,
        items: inventory.allocatedItems,
        address: customer.shippingAddress
      }
    );

    return {
```

```
      orderId,
      status: "completed",
      trackingNumber: shipment.trackingNumber,
      estimatedDelivery: shipment.estimatedDelivery
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

# Main orchestrator function
@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']

    # Step 1: Validate customer by invoking customer service function
    customer = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
        {'customerId': customer_id},
        name='validate-customer'
    )

    if not customer['isValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'invalid_customer'}

    # Step 2: Check inventory by invoking inventory service function
    inventory = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1',
        {'orderId': order_id, 'items': event['items']},
        name='check-inventory'
    )

    if not inventory['available']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'insufficient_inventory'}

    # Step 3: Process payment by invoking payment service function
```

```
    payment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:payment-service:1',
        {
            'customerId': customer_id,
            'amount': inventory['totalAmount'],
            'paymentMethod': customer['paymentMethod']
        },
        name='process-payment'
    )

    # Step 4: Create shipment by invoking fulfillment service function
    shipment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1',
        {
            'orderId': order_id,
            'items': inventory['allocatedItems'],
            'address': customer['shippingAddress']
        },
        name='create-shipment'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': shipment['trackingNumber'],
        'estimatedDelivery': shipment['estimatedDelivery']
    }
```

Jeder Aufruf erstellt einen Checkpoint in der Orchestrator-Funktion. Wenn der Orchestrator nach Abschluss der Kundenvalidierung unterbrochen wird, wird er von diesem Checkpoint aus mit den gespeicherten Kundendaten wieder aufgenommen und der Validierungsaufruf übersprungen. Dies verhindert doppelte Aufrufe von nachgelagerten Diensten und gewährleistet eine konsistente Ausführung auch bei Unterbrechungen.

Bei der aufgerufenen Funktion kann es sich um eine beliebige Lambda-Funktion handeln, einschließlich einer dauerhaften Funktion mit einem eigenen mehrstufigen Workflow, Wartezeiten und Checkpoints. Die aufrufende Funktion wartet auf den Abschluss der dauerhaften Ausführung und erhält dann das Endergebnis.

> **Note**
>
> Cross-account Aufrufe werden nicht unterstützt. Alle aufgerufenen Funktionen müssen sich im selben AWS Konto wie die aufrufende Funktion befinden.

Stapelverarbeitung mit Checkpoints

Verarbeiten Sie Millionen von Datensätzen mit automatischer Wiederherstellung vom letzten erfolgreichen Checkpoint nach Ausfällen. Dieses Beispiel zeigt, wie langlebige Funktionen `map()` Operationen mit Chunking und Ratenbegrenzung kombinieren, um umfangreiche Datenverarbeitungen zu bewältigen.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

interface Batch {
  batchIndex: number;
  recordIds: string[];
}

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize = 1000 } = event;

    // Step 1: Get all record IDs to process
    const recordIds = await context.step("fetch-record-ids", async () => {
      return await dataService.getRecordIds(datasetId);
    });

    // Step 2: Split into batches
    const batches: Batch[] = [];
    for (let i = 0; i < recordIds.length; i += batchSize) {
      batches.push({
        batchIndex: Math.floor(i / batchSize),
        recordIds: recordIds.slice(i, i + batchSize)
      });
    }
```

```
    // Step 3: Process batches with controlled concurrency
    const batchResults = await context.map(
      "process-batches",
      batches,
      async (ctx: DurableContext, batch: Batch, index: number) => {
        const processed = await ctx.step(`batch-${batch.batchIndex}`, async () => {
          const results = [];
          for (const recordId of batch.recordIds) {
            const result = await recordService.process(recordId);
            results.push(result);
          }
          return results;
        });

        const validated = await ctx.step(`validate-${batch.batchIndex}`, async () =>
{
          return await validationService.validateBatch(processed);
        });

        return {
          batchIndex: batch.batchIndex,
          recordCount: batch.recordIds.length,
          successCount: validated.successCount,
          failureCount: validated.failureCount
        };
      },
      {
        maxConcurrency: 5
      }
    );

    // Step 4: Aggregate results
    const summary = await context.step("aggregate-results", async () => {
      const results = batchResults.getResults();
      const totalSuccess = results.reduce((sum, r) => sum + r.successCount, 0);
      const totalFailure = results.reduce((sum, r) => sum + r.failureCount, 0);

      return {
        datasetId,
        totalRecords: recordIds.length,
        batchesProcessed: batches.length,
        successCount: totalSuccess,
        failureCount: totalFailure,
        completedAt: new Date().toISOString()
```

```
      };
    });

    return summary;
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 MapConfig
from datetime import datetime
from typing import List, Dict

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event.get('batchSize', 1000)

    # Step 1: Get all record IDs to process
    record_ids = context.step(
        lambda _: data_service.get_record_ids(dataset_id),
        name='fetch-record-ids'
    )

    # Step 2: Split into batches
    batches = []
    for i in range(0, len(record_ids), batch_size):
        batches.append({
            'batchIndex': i // batch_size,
            'recordIds': record_ids[i:i + batch_size]
        })

    # Step 3: Process batches with controlled concurrency
    def process_batch(ctx: DurableContext, batch: Dict, index: int):
        batch_index = batch['batchIndex']

        def process_records(_):
            results = []
            for record_id in batch['recordIds']:
                result = record_service.process(record_id)
```

```
                results.append(result)
            return results

        processed = ctx.step(process_records, name=f'batch-{batch_index}')

        validated = ctx.step(
            lambda _: validation_service.validate_batch(processed),
            name=f'validate-{batch_index}'
        )

        return {
            'batchIndex': batch_index,
            'recordCount': len(batch['recordIds']),
            'successCount': validated['successCount'],
            'failureCount': validated['failureCount']
        }

    batch_results = context.map(
        process_batch,
        batches,
        name='process-batches',
        config=MapConfig(max_concurrency=5)
    )

    # Step 4: Aggregate results
    def aggregate_results(_):
        results = batch_results.get_results()
        total_success = sum(r['successCount'] for r in results)
        total_failure = sum(r['failureCount'] for r in results)

        return {
            'datasetId': dataset_id,
            'totalRecords': len(record_ids),
            'batchesProcessed': len(batches),
            'successCount': total_success,
            'failureCount': total_failure,
            'completedAt': datetime.now().isoformat()
        }

    summary = context.step(aggregate_results, name='aggregate-results')

    return summary
```

Datensätze werden in überschaubare Stapel aufgeteilt, um eine Überlastung des Speichers oder nachgelagerte Dienste zu vermeiden. Anschließend werden mehrere Batches gleichzeitig verarbeitet, wobei die Parallelität kontrolliert wird. `maxConcurrency` Jeder Stapel hat seinen eigenen Prüfpunkt, sodass bei Fehlern nur der fehlgeschlagene Stapel wiederholt wird, anstatt alle Datensätze erneut zu verarbeiten. Dieses Muster ist ideal für ETL-Jobs, Datenmigrationen oder Massenvorgänge, bei denen die Verarbeitung Stunden dauern kann.

Nächste Schritte

- Machen Sie sich mit grundlegenden Konzepten vertraut DurableContext, lernen Sie Schritte kennen und warten Sie
- Informieren Sie sich über bewährte Methoden zum Schreiben von deterministischem Code und zur Leistungsoptimierung
- Erfahren Sie, wie Sie langlebige Funktionen lokal und in der Cloud testen
- Vergleichen Sie langlebige Funktionen mit Step Functions, um herauszufinden, wann die einzelnen Methoden am effektivsten sind. Siehe Dauerhafte Funktionen oder Step Functions.

Sicherheit und Berechtigungen für langlebige Lambda-Funktionen

AWS Lambda Dauerhafte Funktionen erfordern spezielle IAM-Berechtigungen, um dauerhafte Ausführungen auszuführen und zu verwalten. Folgen Sie dem Prinzip der geringsten Rechte, indem Sie nur die Berechtigungen gewähren, die jeder Prinzipal benötigt.

IAM-Berechtigungen für dauerhafte Funktionen

Für die meisten dauerhaften API-Aktionen autorisiert Lambda die Anforderung anhand des permanenten Ausführungs-ARN, nicht anhand des Funktions-ARN. Eine dauerhafte Ausführung ist eine Unterressource einer Funktionsversion, und ihr ARN erweitert den Funktions-ARN:

`arn:aws:lambda:`*`region`*`:`*`account-id`*`:function:`*`function-name`*`:`*`qualifier`*`/durable-execution/`*`execution-name`*`/`*`execution-id`*

Der Qualifizierer in einer dauerhaften Ausführungs-ARN ist immer `$LATEST` oder eine Versionsnummer.

Wenn Ihre Richtlinie einen unqualifizierten Funktions-ARN im Resource-Element angibt, lehnt Lambda die Anforderung ab, da ein unqualifizierter ARN niemals einem dauerhaften Ausführungs-

ARN entspricht. Um Zugriff zu gewähren, hängen Sie ein Qualifikationsmuster an den Funktions-ARN an:

- `arn:aws:lambda:us-east-1:123456789012:function:my-function:*`— Entspricht dauerhaften Ausführungen aller Versionen.
- `arn:aws:lambda:us-east-1:123456789012:function:my-function:1/durable-execution/*`— Entspricht nur dauerhaften Ausführungen von Version 1.

Die meisten Richtlinien verwenden das `:*` Formular, da sich die Versionsnummern bei jeder Bereitstellung ändern können.

> **Aliase gelten nicht für dauerhafte Ausführungsberechtigungen**
>
> Sie können eine dauerhafte Funktion über einen Alias aufrufen, aber der ARN der resultierenden Ausführung enthält die aufgelöste Versionsnummer.

Berechtigungen für die Ausführungsrolle

Die Ausführungsrolle Ihrer dauerhaften Funktion muss über Berechtigungen zum Erstellen von Checkpoints und zum Abrufen des Ausführungsstatus verfügen. Die folgende Richtlinie zeigt die mindestens erforderlichen Berechtigungen:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name:*"
        }
    ]
}
```

Wenn Sie mithilfe der Konsole eine dauerhafte Funktion erstellen, fügt Lambda diese Berechtigungen automatisch der Ausführungsrolle hinzu. Wenn Sie die Funktion mithilfe von AWS CLI oder erstellen AWS CloudFormation, fügen Sie diese Berechtigungen Ihrer Ausführungsrolle hinzu.

Damit Lambda Ihre Ausführungsrolle übernehmen kann, muss die Vertrauensrichtlinie der Rolle den Lambda-Serviceprinzipal (`lambda.amazonaws.com`) als vertrauenswürdigen Dienst angeben. Das folgende Beispiel zeigt eine Vertrauensrichtlinie, die Lambda die Erlaubnis erteilt, die Rolle zu übernehmen.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Prinzip der geringsten Privilegien

Beschränkt das `Resource` Element auf bestimmte Funktions-ARNs, anstatt Platzhalter zu verwenden. Dies beschränkt die Ausführungsrolle auf Checkpoint-Operationen nur für die Funktionen, die sie benötigen.

Example Eingeschränkte Berechtigungen für mehrere Funktionen

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:123456789012:function:orderProcessor:*",
                "arn:aws:lambda:us-east-1:123456789012:function:paymentHandler:*"
            ]
        }
    ]
}
```

Alternativ können Sie die AWS verwaltete Richtlinie verwenden, die `AWSLambdaBasicDurableExecutionRolePolicy` die erforderlichen dauerhaften Ausführungsberechtigungen zusammen mit den grundlegenden Lambda-Ausführungsberechtigungen für Amazon Logs enthält. CloudWatch

Berechtigungen für die Verwaltung dauerhafter Ausführungen

Lambda autorisiert die `lambda:StopDurableExecution` Aktionen `lambda:GetDurableExecutionlambda:GetDurableExecutionHistory`, und gegen den ARN für die dauerhafte Ausführung.

Um die dauerhaften Ausführungen einer Funktion aufzulisten, verwenden Sie. `lambda:ListDurableExecutionsByFunction` Lambda autorisiert diese Aktion für den Funktions-ARN aus der Anfrage. Die folgenden Regeln gelten für den ARN-Abgleich:

- Eine Richtlinie, die der unqualifizierten Funktion ARN gewährt, entspricht nur Anforderungen, die den unqualifizierten Funktionsnamen verwenden.
- Eine Richtlinie, die einen qualifizierten ARN gewährt, entspricht nur Anforderungen, die einen qualifizierten Funktionsnamen verwenden.

Weitere Informationen finden Sie unter the section called “Referenzierung von Funktionen im Abschnitt „Ressourcen“ von Richtlinien”.

Um Anfragen mit einer beliebigen Form des Funktionsnamens abzugleichen, schließen Sie sowohl den unqualifizierten als auch den qualifizierten Funktions-ARN ein, wie die folgende Richtlinie zeigt:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageDurableExecutions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetDurableExecution",
                "lambda:GetDurableExecutionHistory",
                "lambda:StopDurableExecution"
            ],
            "Resource": "arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:*"
        },
        {
```

```
            "Sid": "ListDurableExecutions",
            "Effect": "Allow",
            "Action": "lambda:ListDurableExecutionsByFunction",
            "Resource": [
                "arn:aws:lambda:us-east-1:123456789012:function:myDurableFunction",
                "arn:aws:lambda:us-east-1:123456789012:function:myDurableFunction:*"
            ]
        }
    ]
}
```

Berechtigungen für das Senden von Rückrufen

Das externe System, das ein Callback-Ergebnis sendet, verwendet die `lambda:SendDurableExecutionCallbackSuccess` Aktionen`lambda:SendDurableExecutionCallbackFailure`, und`lambda:SendDurableExecutionCallbackHeartbeat`.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SendCallbacks",
            "Effect": "Allow",
            "Action": [
                "lambda:SendDurableExecutionCallbackSuccess",
                "lambda:SendDurableExecutionCallbackFailure",
                "lambda:SendDurableExecutionCallbackHeartbeat"
            ],
            "Resource": "arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:*"
        }
    ]
}
```

Lambda autorisiert Rückrufanfragen anhand des ARN der dauerhaften Ausführung, die den Callback generiert hat.

Zustandsverschlüsselung

Dauerhafte Lambda-Funktionen verschlüsseln Ausführungsdaten im Ruhezustand. Bei jeder Funktionsausführung wird ein isolierter Zustand beibehalten, auf den andere Ausführungen nicht

zugreifen können. Sie können einen vom Kunden verwalteten Schlüssel konfigurieren, um dauerhafte Ausführungsdaten zu verschlüsseln. Weitere Informationen finden Sie unter Verschlüsseln AWS Lambda dauerhafte Ausführungsdaten.

Zu den Checkpoint-Daten gehören:

- Ergebnisse und Fehler der Schritte
- Verkettete Invoke-Eingaben

Eine vollständige Liste der dauerhaften Ausführungsdaten, die im Ruhezustand verschlüsselt werden, finden Sie unter. Was ist verschlüsselt

Alle Daten werden während der Übertragung mit TLS verschlüsselt, wenn Lambda Checkpoint-Daten liest oder schreibt.

Benutzerdefinierte Verschlüsselung mit benutzerdefinierten Serializern und Deserializern

Für kritische Sicherheitsanforderungen können Sie mithilfe benutzerdefinierter Serialisierer und Deserialisierer () mithilfe eines dauerhaften SDK Ihren eigenen Verschlüsselungs- und Entschlüsselungsmechanismus implementieren. SerDer Dieser Ansatz gibt Ihnen die volle Kontrolle über die Verschlüsselungsschlüssel und Algorithmen, die zum Schutz der Checkpoint-Daten verwendet werden.

> **Important**
>
> Wenn Sie eine benutzerdefinierte Verschlüsselung verwenden, verlieren Sie die Sichtbarkeit der Betriebsergebnisse in der Lambda-Konsole und der API-Antworten. Checkpoint-Daten werden im Ausführungsverlauf verschlüsselt angezeigt und können ohne Entschlüsselung nicht überprüft werden.

Anforderungen `kms:Encrypt` und `kms:Decrypt` Berechtigungen der Ausführungsrolle Ihrer Funktion für den AWS KMS Schlüssel, der in der benutzerdefinierten SerDer Implementierung verwendet wird.

CloudTrail Protokollierung

Lambda protokolliert Checkpoint-Operationen als Datenereignisse in. AWS CloudTrail Sie können CloudTrail damit überwachen, wann Checkpoints erstellt werden, Änderungen am Ausführungsstatus verfolgen und den Zugriff auf dauerhafte Ausführungsdaten überwachen.

Checkpoint-Operationen werden in CloudTrail Protokollen mit den folgenden Ereignisnamen angezeigt:

- `CheckpointDurableExecution`— Wird protokolliert, wenn ein Schritt abgeschlossen ist und ein Checkpoint erstellt wird
- `GetDurableExecutionState`— Wird protokolliert, wenn Lambda während der Wiedergabe den Ausführungsstatus abruft

Um die Protokollierung von Datenereignissen für dauerhafte Funktionen zu aktivieren, konfigurieren Sie einen CloudTrail Trail zur Protokollierung von Lambda-Datenereignissen. Weitere Informationen finden Sie im CloudTrail Benutzerhandbuch unter Protokollieren von Datenereignissen.

Beispiel: CloudTrail Logeintrag für den Checkpoint-Betrieb

```
{
    "eventVersion": "1.08",
    "eventTime": "2024-11-16T10:30:45Z",
    "eventName": "CheckpointDurableExecution",
    "eventSource": "lambda.amazonaws.com",
    "requestParameters": {
        "functionName": "myDurableFunction",
        "executionId": "exec-abc123",
        "stepId": "step-1"
    },
    "responseElements": null,
    "eventType": "AwsApiCall"
}
```

Cross-account Überlegungen

Wenn Sie dauerhafte Funktionen kontoübergreifend AWS aufrufen, benötigt das aufrufende Konto eine `lambda:InvokeFunction` Genehmigung, Checkpoint-Operationen verwenden jedoch immer die Ausführungsrolle im Konto der Funktion. Das aufrufende Konto kann nicht direkt auf Checkpoint-Daten oder den Ausführungsstatus zugreifen.

Diese Isolierung stellt sicher, dass die Checkpoint-Daten im Konto der Funktion sicher bleiben, auch wenn sie von externen Konten aus aufgerufen werden.

Übernommene Lambda-Sicherheitsfunktionen

Dauerhafte Funktionen übernehmen alle Sicherheits-, Governance- und Compliance-Funktionen von Lambda. Dazu gehören VPC-Konnektivität, Verschlüsselung mit Umgebungsvariablen, Warteschlangen, reservierte Parallelität, Funktions-URLs, Codesignaturen und Compliance-Zertifizierungen wie SOC, PCI DSS und HIPAA.

Ausführliche Informationen zu den Lambda-Sicherheitsfunktionen finden Sie unter Sicherheit im Lambda Developer Guide. AWS Lambda Die einzigen zusätzlichen Sicherheitsaspekte für langlebige Funktionen sind die in diesem Handbuch dokumentierten IAM-Berechtigungen.

Verschlüsseln AWS Lambda dauerhafte Ausführungsdaten

Mit AWS Lambda langlebigen Funktionen können Sie robuste, mehrstufige Anwendungen erstellen, die bis zu einem Jahr lang ausgeführt werden können. Mithilfe von Checkpoints können Sie jede Ausführung nach Ausfällen wiederherstellen, indem abgeschlossene Arbeiten erneut abgespielt werden.

Lambda verschlüsselt dauerhafte Ausführungsdaten immer im Ruhezustand. Sie können zusätzlich Ihren eigenen, vom AWS KMS Kunden verwalteten Schlüssel für die Funktion zur Rotationskontrolle, Audittransparenz oder Einhaltung von Vorschriften konfigurieren. Bei einem vom Kunden verwalteten Schlüssel können nur von Ihnen autorisierte Benutzer Ausführungsdaten lesen.

Wie funktioniert Verschlüsselung

Eine dauerhafte Lambda-Ausführung verwendet während ihrer gesamten Lebensdauer denselben KMS-Schlüssel, mit dem sie begonnen hat. Das Ändern oder Entfernen des Schlüssels für die Funktion wirkt sich nur auf Ausführungen aus, die nach der Änderung beginnen. Erfahren SieWenn der vom Kunden verwaltete Schlüssel nicht verfügbar ist, wie sich eine dauerhafte Ausführung verhält, wenn ihr Schlüssel nicht verfügbar ist.

Für vom Kunden verwaltete Schlüssel fallen AWS KMS Standardgebühren an. Details zu den Preisen finden Sie unter AWS Key Management Service -Preise.

Was ist verschlüsselt

Wenn Sie einen vom Kunden verwalteten Schlüssel für eine dauerhafte Funktion konfigurieren, verwendet Lambda diesen Schlüssel, um die folgenden dauerhaften Ausführungsdaten im Ruhezustand zu verschlüsseln:

- Die Eingabe-Payload, die Sie mit jeder Anfrage übergeben. `Invoke`
- Checkpoint-Daten, die von der `CheckpointDurableExecution` API dauerhaft gespeichert werden, einschließlich Schrittergebnisse, Schrittfehler und verkettete Aufrufeingaben.
- Ausführungsergebnisse und Fehler.
- Rückrufergebnisse und Fehler, die Sie über `SendDurableExecutionCallbackSuccess` und `SendDurableExecutionCallbackFailure` einreichen.

> **Function-level und langlebige Ausführungsschlüssel sind unabhängig**
>
> Die Funktionsebene`KMSKeyArn`, die Umgebungsvariablen, ZIP-Bereitstellungspakete und SnapStartSnapshots verschlüsselt, unterscheidet sich von der Funktionsebene, die dauerhafte `KMSKeyArn` `DurableConfig` Ausführungsdaten verschlüsselt. Wenn Sie das eine festlegen, wird das andere nicht festgelegt. Sie können denselben KMS-Schlüssel für beide verwenden, oder Sie können unterschiedliche KMS-Schlüssel verwenden.

Richten Sie die vom Kunden verwaltete Schlüsselverschlüsselung ein

Die Einrichtung der vom Kunden verwalteten Schlüsselverschlüsselung für eine dauerhafte Funktion erfolgt in drei Schritten. Führen Sie die folgenden Schritte der Reihe nach aus.

Erstellen eines kundenseitig verwalteten Schlüssels

Dauerhafte Funktionen unterstützen KMS-Schlüssel mit symmetrischer Verschlüsselung in derselben AWS Region wie die Funktion. Cross-Region Schlüssel werden nicht unterstützt. Um einen symmetrischen, vom Kunden verwalteten Schlüssel zu erstellen, folgen Sie den Schritten zur Erstellung von KMS-Schlüsseln mit symmetrischer Verschlüsselung im AWS Key Management Service Entwicklerhandbuch.

Berechtigungen

Sie gewähren AWS KMS Berechtigungen über die Schlüsselrichtlinie des KMS-Schlüssels.

Für die Konfiguration eines vom Kunden verwalteten Schlüssels sind AWS KMS Berechtigungen für den Prinzipal erforderlich, der die Funktion erstellt oder aktualisiert. Für das Aufrufen einer dauerhaften Funktion sind keine AWS KMS Berechtigungen für den Aufrufer erforderlich. Lambda führt die Verschlüsselung mit seinem Dienstprinzipal durch.

Schlüsselrichtlinie

Schlüsselrichtlinien steuern den Zugriff auf den vom Kunden verwalteten Schlüssel. Jeder vom Kunden verwaltete Schlüssel muss genau eine Schlüsselrichtlinie haben. `Resource: "*"`Bezieht sich in einer Schlüsselrichtlinie nur auf den KMS-Schlüssel, mit dem die Richtlinie verknüpft ist, nicht auf alle KMS-Schlüssel in Ihrem Konto.

Die Schlüsselrichtlinie einer dauerhaften Funktion gewährt jedem Principal nur die AWS KMS Aktionen, die er benötigt, aufgeteilt nach Dienst- und Funktions-ARN. In den folgenden Abschnitten werden die Aussagen und Bedingungen sowie ein vollständiges Beispiel beschrieben.

Erforderliche Richtlinienerklärungen

Die Richtlinie gewährt die folgenden Funktionen:

- Aktivieren Sie IAM-Benutzerberechtigungen. Gewährt dem Kontostamm bedingungslosen Zugriff zur Verwaltung des Schlüssels. Diese Anweisung verwendet keine Bedingungen, da eine Festlegung des Gültigkeitsbereichs Sie daran hindern würde, den Schlüssel zu rotieren, zu aktualisieren oder zu löschen.
- Erlauben Sie Lambda, diesen Schlüssel für dauerhafte Funktionen zu verwenden. Gewährt dem Lambda-Service Principal `kms:GenerateDataKey` und`kms:Decrypt`.
- Erlaubt der Funktionsausführungsrolle, dauerhafte Ausführungsdaten zu entschlüsseln. Erteilt der Ausführungsrolle `kms:Decrypt` das Lesen des Status und das Fortschreiten der Ausführung.
- Erlaubt dem Autor der Funktion, diesen Schlüssel zu beschreiben. Zuschüsse`kms:DescribeKey`, damit Lambda überprüfen kann, ob der Schlüssel symmetrisch ist und während `CreateFunction` oder aktiviert ist. `UpdateFunctionConfiguration`
- Erlaubt dem Autor der Funktion, diesen Schlüssel für eine bestimmte Funktion zu validieren. Grants `kms:GenerateDataKey` und `kms:Decrypt` so kann Lambda wichtige Berechtigungen und Zugriffe mit dem Verschlüsselungskontext der Funktion während `CreateFunction` oder `UpdateFunctionConfiguration` validieren. Dadurch wird bestätigt, dass die Schlüsselrichtlinie Aufrufe für diese spezielle Funktion akzeptiert, bevor die Funktion erstellt oder aktualisiert wird.

- Erlaubt dauerhafte Ausführungsoperatoren. Erteilt Operatoren `kms:Decrypt` für Aufrufe von `GetDurableExecutionIncludeExecutionData=true`, `GetDurableExecutionHistory` mit `GetDurableExecutionStateStopDurableExecution`, und den Callback-APIs.

Es hat sich bewährt, separate Prinzipale für die Ausführungsrolle, den Funktionsautor und den dauerhaften Ausführungsoperator zu verwenden, sodass jede Identität nur über die Funktionen verfügt, die sie benötigt.

Empfohlene politische Bedingungen

In der Richtlinie werden die folgenden Bedingungen verwendet, um den Zugriff einzuschränken:

- kms: ViaService schränkt die Verwendung von Schlüsseln auf Anfragen ein, die über Lambda weitergeleitet werden. Wenn dies auf die Anweisungen der Ausführungsrolle, des Funktionsautors und des Operators angewendet wird, wird verhindert, dass diese den Schlüssel direkt verwenden. AWS KMS
- aws: SourceArn und aws: SourceAccount schützen vor dem dienstübergreifenden Problem mit verwirrten Stellvertretern. Lambda leitet diese Werte weiter, wenn es AWS KMS mit seinen eigenen Dienstanmeldedaten aufruft. Diese Bedingung gilt nur für die Lambda Service Principal Statement. Die Anweisungen für die Ausführungsrolle, den Autor der Funktion und den Operator werden AWS KMS mit den Anmeldeinformationen für die Forward-Access-Sitzung des Aufrufers aufgerufen, die diese Header nicht enthalten.
- kms: Der Schlüssel ist EncryptionContext:aws:lambda:FunctionArn auf eine bestimmte Funktion beschränkt. Lambda fügt dies dem Verschlüsselungskontext bei jedem AWS KMS Aufruf für dauerhafte Ausführungsdaten hinzu.

Das folgende Beispiel kombiniert die obigen Anweisungen und Bedingungen.

Example Wichtige Richtlinie für langlebige Funktionen

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Enable IAM User Permissions",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::111122223333:root" },
            "Action": "kms:*",
            "Resource": "*"
```

```
        },
        {
            "Sid": "Allow Lambda to use this key for durable functions",
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": [ "kms:GenerateDataKey", "kms:Decrypt" ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "111122223333",
                    "aws:SourceArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction",
                    "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction"
                }
            }
        },
        {
            "Sid": "Allow the function execution role to decrypt durable execution
 data",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::111122223333:role/
myDurableFunctionRole" },
            "Action": "kms:Decrypt",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com",
                    "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction"
                }
            }
        },
        {
            "Sid": "Allow the function author to describe this key",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::111122223333:role/FunctionAuthor" },
            "Action": "kms:DescribeKey",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
```

```
        },
        {
            "Sid": "Allow the function author to validate this key for a specific
 function",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::111122223333:role/FunctionAuthor" },
            "Action": [ "kms:GenerateDataKey", "kms:Decrypt" ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com",
                    "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction"
                }
            }
        },
        {
            "Sid": "Allow durable execution operators",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::111122223333:role/
DurableExecutionOperator" },
            "Action": "kms:Decrypt",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com",
                    "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction"
                }
            }
        }
    ]
}
```

Weitere Informationen zu AWS KMS wichtigen Richtlinien finden Sie unter So ändern Sie eine wichtige Richtlinie im AWS Key Management Service Entwicklerhandbuch.

Konfigurieren Sie einen vom Kunden verwalteten Schlüssel für eine dauerhafte Funktion

Sie konfigurieren den vom Kunden verwalteten Schlüssel für dauerhafte Ausführungsdaten über das `KMSKeyArn` Feld des `DurableConfig` Objekts. Legen Sie ihn fest, wenn Sie die Funktion mit erstellen CreateFunction; aktualisieren Sie ihn bei einer vorhandenen dauerhaften Funktion mit UpdateFunctionConfiguration.

Lambda console

Um einen vom Kunden verwalteten Schlüssel für eine bestehende dauerhafte Funktion zu konfigurieren

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine dauerhafte Funktion.
3. Wählen Sie „Konfiguration“ und anschließend in der linken Navigationsleiste „Dauerhafte Ausführung“.
4. Wählen Sie Bearbeiten aus.
5. Wählen Sie unter Verschlüsselung die Option Einen vom Kunden verwalteten Schlüssel verwenden und wählen Sie dann einen KMS-Schlüssel aus der Liste aus.
6. Wählen Sie Speichern.

AWS CLI

Um einen vom Kunden verwalteten Schlüssel zu konfigurieren, wenn Sie eine Funktion erstellen

Im folgenden Beispiel für eine Funktion zum Erstellen einer Funktion gibt die `--durable-config` Option den vom Kunden verwalteten Schlüssel-ARN zusammen mit dem dauerhaften Ausführungstimeout und der Aufbewahrungsdauer an.

```
aws lambda create-function \
  --function-name myDurableFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --role arn:aws:iam::111122223333:role/myDurableFunctionRole \
  --zip-file fileb://function.zip \
  --durable-config '{"KMSKeyArn":"arn:aws:kms:us-east-1:111122223333:key/key-id","ExecutionTimeout":3600,"RetentionPeriodInDays":30}'
```

Um einen vom Kunden verwalteten Schlüssel für eine bestehende dauerhafte Funktion zu konfigurieren

Verwenden Sie den Befehl update-function-configuration. Schließt alle `DurableConfig` Felder ein, die Sie behalten möchten, da das gesamte Objekt `--durable-config` ersetzt wird.

```
aws lambda update-function-configuration \
```

```
  --function-name myDurableFunction \
  --durable-config '{"KMSKeyArn":"arn:aws:kms:us-east-1:111122223333:key/key-
id","ExecutionTimeout":3600,"RetentionPeriodInDays":30}'
```

Um den vom Kunden verwalteten Schlüssel aus einer dauerhaften Funktion zu entfernen

Auslassen `KMSKeyArn` von`--durable-config`. Bestehende Ausführungen verwenden weiterhin den vom Kunden verwalteten Schlüssel, mit dem sie begonnen haben. Neue Ausführungen verwenden stattdessen die Standardverschlüsselung.

```
aws lambda update-function-configuration \
  --function-name myDurableFunction \
  --durable-config '{"ExecutionTimeout":3600,"RetentionPeriodInDays":30}'
```

AWS CloudFormation

Legen Sie in einer `AWS::Lambda::Function` Ressource die `KMSKeyArn` Eigenschaft fest: `DurableConfig`

```
Resources:
  MyDurableFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs24.x
      Handler: index.handler
      Role: !GetAtt MyDurableFunctionRole.Arn
      Code:
        ZipFile: |
          // Your durable function code
      DurableConfig:
        KMSKeyArn: "arn:aws:kms:us-east-1:111122223333:key/key-id"
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 30
```

Important

Jede dauerhafte Ausführung verwendet den vom Kunden verwalteten Schlüssel, der beim Start für die Funktion konfiguriert wurde. Das Ändern oder Entfernen des Schlüssels wirkt sich nur auf Ausführungen aus, die nach der Änderung beginnen. Bei laufenden

Ausführungen wird weiterhin der Schlüssel verwendet, mit dem sie begonnen haben, bis sie abgeschlossen sind oder fehlschlagen.

Dauerhafte Ausführungsdaten werden gelesen

Zwei Lese-APIs geben dauerhafte Ausführungsdaten zurück:

- `GetDurableExecution`gibt den aktuellen Status einer einzelnen Ausführung zurück, einschließlich der eingegebenen Nutzdaten, der neuesten Checkpoint-Daten und der Ergebnis- oder Fehlerinformationen.
- `GetDurableExecutionHistory`gibt die geordnete Liste der Ereignisse zurück, die von der Ausführung ausgelöst wurden, und zeigt Checkpoint-Eingaben und -Ergebnisse, Chained-Invoke-Eingaben und -Ergebnisse sowie das Endergebnis an.

Beide APIs akzeptieren einen Anforderungsparameter. `IncludeExecutionData` Wenn `IncludeExecutionData` ja`true`, verwendet Lambda die Anmeldeinformationen des Anrufers, um den vom Kunden verwalteten Schlüssel `kms:Decrypt` aufzurufen. Lambda gibt dann die entschlüsselten Ausführungsdaten in der Antwort zurück. Die Identität des Anrufers muss `kms:Decrypt` auf dem vom Kunden verwalteten Schlüssel stehen.

Wenn dies `IncludeExecutionData` der `false` Fall ist, ruft Lambda keine Ausführungsdaten auf AWS KMS oder entschlüsselt sie nicht, und die Antwort wird auf gesetzt`ExecutionDataIncluded`. `false` Die Antwort enthält immer noch`DurableConfig`, was den vom Kunden verwalteten Schlüssel-ARN wiedergibt, den die Ausführung verwendet hat. `IncludeExecutionData`ist standardmäßig auf eingestellt`false`, sodass Anrufer, die nur Metadaten benötigen, keine Berechtigungen benötigen. AWS KMS

Beispiel: **`GetDurableExecution`** Antwort mit **`IncludeExecutionData=false`**

```
{
    "DurableExecutionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction:execution:exec-abc123",
    "DurableExecutionName": "exec-abc123",
    "FunctionArn": "arn:aws:lambda:us-east-1:111122223333:function:myDurableFunction",
    "StartTimestamp": 1733256000,
    "Status": "RUNNING",
    "ExecutionDataIncluded": false,
    "DurableConfig": {
```

```
        "ExecutionTimeout": 3600,
        "KMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id",
        "RetentionPeriodInDays": 30
    }
}
```

Chained ruft auf

Wenn eine dauerhafte Funktion Chained Invoke verwendet, um eine andere dauerhafte Funktion aufzurufen, behandelt Lambda die untergeordnete Ausführung als unabhängige dauerhafte Ausführung: Sie hat ihre eigene Ausführungs-ID, ihren eigenen Checkpoint-Stream und ihren eigenen vom Kunden verwalteten Schlüssel. Die vom Kunden verwaltete Schlüsselkonfiguration des Elternteils hat keine Auswirkungen auf die Daten des Kindes, und die vom Kunden verwaltete Schlüsselkonfiguration des Kindes hat keine Auswirkungen auf die Daten des Elternteils.

Berechtigungen

Der Lambda-Service Principal muss über `kms:GenerateDataKey` und `kms:Decrypt` auf dem vom Kunden verwalteten Schlüssel der untergeordneten Funktion verfügen. Wenn die übergeordnete und die untergeordnete Funktion unterschiedliche vom Kunden verwaltete Schlüssel verwenden, gewähren Sie dem Dienstprinzipal Zugriff auf jeden Schlüssel separat. Für die Ausführungsrolle der übergeordneten Funktion sind keine Berechtigungen für den vom Kunden verwalteten Schlüssel des untergeordneten Elements erforderlich.

Identifizieren Sie, welcher Schlüssel welche Ausführung verschlüsselt hat

Beides `GetDurableExecution` und `ListDurableExecutionsByFunction` geben den vom Kunden verwalteten Schlüssel-ARN zurück, der jede Ausführung verschlüsselt hat. Verwenden Sie diese APIs, um zu überprüfen, welcher Schlüssel gültig war, als eine übergeordnete oder untergeordnete Ausführung gestartet wurde.

Datenschlüssel-Caching

Um das AWS KMS Anrufvolumen zu reduzieren und die Verfügbarkeit bei Serviceunterbrechungen zu verbessern, speichert Lambda einen Datenschlüssel, der aus Ihrem vom Kunden verwalteten Schlüssel generiert wurde, für bis zu 15 Minuten im Cache. Solange der Cache warm ist, verwendet Lambda denselben Datenschlüssel für alle Operationen innerhalb einer Ausführung und für alle Ausführungen, die während des Cache-Fensters gestartet wurden.

Cost (Kosten)

Per-execution AWS KMS Die Kosten bleiben niedrig, da Lambda `GenerateDataKey` höchstens einmal pro Cache-Fenster aufruft, nicht einmal pro Checkpoint. Bei hohen Anforderungsraten sinken die Kosten pro Ausführung weiter, da mehr Ausführungen jeden zwischengespeicherten Datenschlüssel gemeinsam nutzen. Ihnen werden die AWS KMS Aufrufe in Rechnung gestellt, die Lambda tatsächlich tätigt, nicht für jeden Checkpoint oder jeden Lesevorgang.

Statische Stabilität

Datenschlüssel bleiben bis zu 15 Minuten zwischengespeichert. Änderungen an Ihrem Schlüssel werden bei der nächsten Cache-Aktualisierung wirksam. Bei längeren Serviceunterbrechungen bedient Lambda weiterhin den Service aus dem Cache, sodass die Ausführung während des Fluges weiterläuft.

CloudTrail

Das Zwischenspeichern von Datenschlüsseln bedeutet, dass Sie weniger `GenerateDataKey` Ereignisse sehen, CloudTrail als es Ausführungen oder Checkpoints gibt. Sehen Sie sich Überwachung von KMS-Schlüsseln im Hinblick auf dauerhafte Funktionen zum Beispiel Ereignisse an.

Wenn der vom Kunden verwaltete Schlüssel nicht verfügbar ist

Wenn der vom Kunden verwaltete Schlüssel, mit dem eine Ausführung gestartet wurde, deaktiviert ist, zum Löschen geplant ist oder sein Zugriff aufgrund der Schlüsselrichtlinie gesperrt wurde, kann die Ausführung nicht weiter voranschreiten. Beim nächsten Checkpoint schlägt Lambda die Ausführung mit einem Fehler fehl, der nicht erneut versucht werden kann. AWS KMS Durch die Wiederherstellung des Zugriffs auf den Schlüssel wird die Ausführung nicht automatisch fortgesetzt. Sie müssen eine neue Ausführung starten.

APIs, die Payloads lesen oder schreiben, schlagen ebenfalls fehl, wenn der Schlüssel nicht verfügbar ist. Sie können eine der folgenden Ausnahmen zurückgeben:

- `KMSAccessDeniedException`: Lambda konnte dauerhafte Ausführungsdaten nicht entschlüsseln, da der Zugriff auf den KMS-Schlüssel verweigert wurde.
- `KMSDisabledException`: Lambda konnte dauerhafte Ausführungsdaten nicht entschlüsseln, da der KMS-Schlüssel deaktiviert ist.
- `KMSInvalidStateException`: Lambda konnte dauerhafte Ausführungsdaten nicht entschlüsseln, da der Status des KMS-Schlüssels nicht gültig ist für. `Decrypt`

- `KMSNotFoundException`: Lambda konnte dauerhafte Ausführungsdaten nicht entschlüsseln, da der KMS-Schlüssel nicht gefunden wurde.

Durch die Wiederherstellung des Zugriffs auf den KMS-Schlüssel können diese Lese-APIs bei Ausführungen, die bereits fehlgeschlagen sind, wieder erfolgreich sein. Fehlgeschlagene Ausführungen bleiben fehlschlagen. Sie müssen eine neue Ausführung starten.

Um die Ausführungsmetadaten zu überprüfen, während der KMS-Schlüssel nicht verfügbar ist, rufen Sie `GetDurableExecution` mit `IncludeExecutionData=false` an. Dadurch werden der Status, die Zeitstempel und `DurableConfig` (einschließlich des KMS-Schlüssels ARN) der Ausführung ohne Aufruf AWS KMS zurückgegeben.

Da Lambda Datenschlüssel zwischenspeichert, wird die Deaktivierung eines Schlüssels nicht sofort wirksam. Details hierzu finden Sie unter Datenschlüssel-Caching.

> **Important**
>
> Durch das Löschen eines KMS-Schlüssels, von dem laufende oder beibehaltene Ausführungen immer noch abhängen, werden diese Ausführungen und ihr Verlauf dauerhaft zerstört.

Überwachung von KMS-Schlüsseln im Hinblick auf dauerhafte Funktionen

Wenn Sie einen vom Kunden verwalteten Schlüssel mit einer dauerhaften Funktion verwenden, können Sie AWS CloudTraildamit die AWS KMS Anrufe verfolgen, die Lambda in Ihrem Namen tätigt. Allgemeine Hinweise zur Suche nach AWS KMS Ereignissen finden Sie unter Suchen nach AWS KMS API-Aktivitäten.

Um sicherzustellen, dass Lambda Ihren Schlüssel wie vorgesehen verwendet, suchen Sie in jedem Ereignis nach diesen Feldern:

- `eventName`: einer von `GenerateDataKeyDecrypt`, oder `DescribeKey`
- `eventSource`: `kms.amazonaws.com`
- `userIdentity.invokedBy`: `lambda.amazonaws.com`
- `requestParameters.encryptionContext.aws:lambda:FunctionArn`: der ARN der dauerhaften Funktion

Bei den folgenden Beispielen handelt es sich um CloudTrail Ereignisse aus einem `CreateFunction` `UpdateFunctionConfiguration` OR-Aufruf, bei dem ein vom Kunden verwalteter Schlüssel für eine dauerhafte Funktion konfiguriert wird. Lambda gibt drei AWS KMS Aufrufe aus, um den Schlüssel zu validieren: einen echten und einen `DescribeKey` Probelauf und. `GenerateDataKey` `Decrypt`

GenerateDataKey

Wenn Sie die `KMSKeyArn` Eingabe festlegen oder ändern`DurableConfig`, führt Lambda einen Probelauf mit Ihrem vom `GenerateDataKey` Kunden verwalteten Schlüssel aus, um zu überprüfen, ob die Schlüsselrichtlinie es Lambda ermöglicht, Datenschlüssel für den Verschlüsselungskontext dieser Funktion abzuleiten. Der Testlauf führt keine kryptografische Arbeit durch, sondern zeichnet ein vollständiges Ereignis auf. CloudTrail Das folgende Beispielereignis zeichnet den Probelauf auf: `GenerateDataKey`


```
{
    "eventVersion": "1.11",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::111122223333:assumed-role/FunctionAuthor/example",
        "accountId": "111122223333",
        "accessKeyId": "ASIA123456789EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/FunctionAuthor",
                "accountId": "111122223333",
                "userName": "FunctionAuthor"
            },
            "attributes": {
                "creationDate": "2026-01-15T16:54:42Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2026-01-15T16:55:00Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-east-1",
```

```
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "errorCode": "DryRunOperationException",
    "errorMessage": "The request would have succeeded, but the DryRun option is
 set.",
    "requestParameters": {
        "encryptionContext": {
            "aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction"
        },
        "dryRun": true,
        "keyId": "arn:aws:kms:us-east-1:111122223333:key/key-id",
        "keySpec": "AES_256"
    },
    "responseElements": null,
    "additionalEventData": {
        "keyMaterialId":
 "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0EXAMPLE"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-east-1:111122223333:key/key-id"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```


Decrypt

Wenn Sie die `KMSKeyArn` Eingabe festlegen oder ändern`DurableConfig`, führt `Decrypt` Lambda auch einen Probelauf mit Ihrem vom Kunden verwalteten Schlüssel aus, um zu überprüfen, ob die Schlüsselrichtlinie es Lambda ermöglicht, Daten für den Verschlüsselungskontext dieser Funktion zu entschlüsseln. Der Testlauf verwendet, sodass kein echter Chiffretext erforderlich `IGNORE_CIPHERTEXT` ist. Das folgende Beispielereignis zeichnet den Probelauf auf: `Decrypt`

```
{
    "eventVersion": "1.11",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::111122223333:assumed-role/FunctionAuthor/example",
        "accountId": "111122223333",
        "accessKeyId": "ASIA123456789EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/FunctionAuthor",
                "accountId": "111122223333",
                "userName": "FunctionAuthor"
            },
            "attributes": {
                "creationDate": "2026-01-15T16:54:42Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2026-01-15T16:55:00Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "errorCode": "DryRunOperationException",
    "errorMessage": "The request would have succeeded, but the DryRun option is
 set.",
    "requestParameters": {
        "encryptionContext": {
            "aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:111122223333:function:myDurableFunction"
        },
        "dryRun": true,
        "keyId": "arn:aws:kms:us-east-1:111122223333:key/key-id",
        "dryRunModifiers": ["IGNORE_CIPHERTEXT"],
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
```

```
    "additionalEventData": {
        "keyMaterialId":
 "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0EXAMPLE"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-east-1:111122223333:key/key-id"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

DescribeKey

Wenn Sie das KMSKeyArn In setzen oder ändernDurableConfig, ruft Lambda auf, DescribeKey um zu bestätigen, dass der Schlüssel symmetrisch und aktiviert ist, bevor es die Änderung akzeptiert. Im Gegensatz zu den Decrypt Sonden GenerateDataKey und ist dies ein echter Aufruf, kein Probelauf. Das folgende Beispielereignis zeichnet den Vorgang DescribeKey auf:

```
{
    "eventVersion": "1.11",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::111122223333:assumed-role/FunctionAuthor/example",
        "accountId": "111122223333",
        "accessKeyId": "ASIA123456789EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/FunctionAuthor",
                "accountId": "111122223333",
                "userName": "FunctionAuthor"
```

```
            },
            "attributes": {
                "creationDate": "2026-01-15T16:54:42Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2026-01-15T16:55:00Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:us-east-1:111122223333:key/key-id"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-east-1:111122223333:key/key-id"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```


Verwandte Verschlüsselungsthemen

- Datenverschlüsselung im Ruhezustand für Lambda
- Sicherung von Lambda-Umgebungsvariablen
- Verschlüsselung von Lambda-ZIP-Bereitstellungspaketen
- Sicherheitsmodell für Lambda SnapStart
- Dauerhafte Lambda-Funktionen konfigurieren

SDK für langlebige Ausführung

Das Durable Execution SDK ist die Grundlage für die Entwicklung langlebiger Funktionen. Es bietet Grundfunktionen, um den Fortschritt zu überprüfen, Wiederholungsversuche zu verarbeiten und den Ausführungsablauf zu verwalten. Das SDK abstrahiert die Komplexität der Checkpoint-Verwaltung und -Wiedergabe und ermöglicht es Ihnen, sequenziellen Code zu schreiben, der automatisch fehlertolerant wird.

Das SDK ist verfügbar für JavaScript, TypeScript, Python, Java, Go und C# (.NET). Eine vollständige API-Dokumentation, Schnellstart-Tutorials und sprachspezifische Anleitungen finden Sie im AWS Durable Execution SDK — Entwicklerhandbuch.

Was macht das SDK

Verwaltung von Checkpoints: Das SDK erstellt automatisch Checkpoints, wenn Ihre Funktion dauerhafte Operationen ausführt. Jeder Checkpoint zeichnet den Operationstyp, die Eingaben und die Ergebnisse auf. Wenn Ihre Funktion einen Schritt abschließt, behält das SDK den Checkpoint bei, bevor es fortfährt. Dadurch wird sichergestellt, dass Ihre Funktion nach jedem abgeschlossenen Vorgang wieder aufgenommen werden kann, falls er unterbrochen wird.

Koordination der Wiedergabe: Wenn Ihre Funktion nach einer Pause oder Unterbrechung wieder aufgenommen wird, führt das SDK eine Wiederholung durch. Es führt Ihren Code von Anfang an aus, überspringt jedoch abgeschlossene Operationen und verwendet gespeicherte Checkpoint-Ergebnisse, anstatt sie erneut auszuführen. Das SDK stellt sicher, dass die Wiedergabe deterministisch ist. Bei gleichen Eingaben und demselben Checkpoint-Log liefert Ihre Funktion dieselben Ergebnisse.

Staatliche Isolierung: Das SDK verwaltet den Ausführungsstatus getrennt von Ihrer Geschäftslogik. Jede dauerhafte Ausführung hat ihr eigenes Checkpoint-Log, auf das andere Ausführungen nicht zugreifen können. Das SDK verschlüsselt Checkpoint-Daten im Ruhezustand und stellt sicher, dass der Status bei allen Wiederholungen konsistent bleibt.

Eine ausführliche Erläuterung der Funktionsweise von Checkpointing und des Wiedergabeverhaltens finden Sie unter Die wichtigsten Konzepte in der AWS Durable Execution SDK-Entwicklerhandbuch.

Dauerhafte Vorgänge

Das SDK bietet Ihrer Funktion eine `DurableContext` Objekt. Dieser Kontext ersetzt den Standard-Lambda-Kontext und bietet Methoden zum Erstellen von Prüfpunkten, zum Verwalten des Ausführungsablaufs und zur Koordination mit externen Systemen.

Der `DurableContext` bietet die folgenden Operationen zum Erstellen dauerhafter Arbeitsabläufe:

Operation	Description
Schritt	Führen Sie eine Arbeitseinheit mit konfigurierbaren Wiederhol ungsstrategien und Ausführungssemantik aus und überprüfen Sie sie.
Wait	Unterbrechen Sie die Ausführung für eine bestimmte Dauer, ohne Rechenressourcen zu verbrauchen.
Warte auf die Bedingung	Abfrage einer Bedingung mit automatischem Checkpoint zwischen den Versuchen.
Rückruf	Unterbrechen Sie die Ausführung und warten Sie, bis ein externes System Eingaben über die Lambda-API bereitstellt.
Invoke	Rufen Sie eine weitere Lambda-Funktion auf und warten Sie mit automatischem Checkpointing auf ihr Ergebnis.
Parallel	Führen Sie mehrere Operationen gleichzeitig mit konfigurierbaren Abschlussrichtlinien aus.
Zuordnung	Verarbeiten Sie jedes Element in einer Sammlung gleichzeitig mit optionaler Parallelitätssteuerung.
Untergeordneter Kontext	Erstellen Sie einen isolierten Ausführungskontext zum Gruppieren mehrerer Operationen.

Bei jeder dauerhaften Operation werden automatisch Checkpoints erstellt, wodurch sichergestellt wird, dass Ihre Funktion von jedem Punkt aus wieder aufgenommen werden kann. Eine ausführliche API-Referenz, Codebeispiele und eine sprachspezifische Verwendung finden Sie unter SDK-Referenz in der AWS Durable Execution SDK-Entwicklerhandbuch.

Wie werden langlebige Betriebsabläufe gemessen

Jede dauerhafte Operation, die Sie aufrufen `DurableContext` erstellt Checkpoints, um den Ausführungsfortschritt zu verfolgen und Statusdaten zu speichern. Für diese Vorgänge fallen je nach Nutzung Gebühren an, und die Checkpoints können Daten enthalten, die zu Ihren Schreib- und

Aufbewahrungskosten beitragen. Zu den gespeicherten Daten gehören Daten zu Aufrufereignissen, Payloads, die von Schritten zurückgegeben werden, und Daten, die beim Abschließen von Rückrufen weitergegeben werden. Wenn Sie wissen, wie langlebige Betriebsabläufe gemessen werden, können Sie die Ausführungskosten abschätzen und Ihre Arbeitsabläufe optimieren. Einzelheiten zur Preisgestaltung finden Sie in der Lambda-Preisseite.

Die Nutzlastgröße bezieht sich auf die Größe der serialisierten Daten, die bei einem dauerhaften Vorgang bestehen bleiben. Die Daten werden in Byte gemessen und die Größe kann je nach dem für die Operation verwendeten Serializer variieren. Die Nutzlast einer Operation kann das Ergebnis selbst für erfolgreiche Abschlüsse sein, oder das serialisierte Fehlerobjekt, wenn die Operation fehlgeschlagen ist.

Grundlegende Vorgänge

Grundlegende Operationen sind die grundlegenden Bausteine für dauerhafte Funktionen:

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Daten wurden beibehalten
Ausführung	Gestartet	1	Größe der Eingabe-Nutzlast
Ausführung	Abgeschlossen (Succeeded/Failed/Stopped)	0	Größe der Ausgangsnutzlast
Schritt	Retry/Succeeded/Failed	10 + Keine Wiederholungen	Bei jedem Versuch wurde die Nutzdatengröße zurückgegeben
Wait	Gestartet	1	N/A
WaitForCondition	Bei jedem Versuch einer Umfrage	+1 N Umfragen	Bei jedem Abfrageversuch wurde die Nutzdatengröße zurückgegeben

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Daten wurden beibehalten
Invocation-level Versuchen Sie es erneut	Gestartet	1	Nutzlast für Fehlerobjekt

Callback-Operationen

Callback-Operationen ermöglichen es Ihrer Funktion, anzuhalten und darauf zu warten, dass externe Systeme Eingaben bereitstellen. Diese Operationen erzeugen Checkpoints, wenn der Callback erstellt wird und wenn er abgeschlossen ist:

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Daten wurden beibehalten
CreateCallback	Gestartet	1	N/A
Abschluss des Callbacks durch API-Aufruf	Completed	0	Nutzlast für den Rückruf
WaitForCallback	Gestartet	3 + Keine Wiederholungen (Kontext + Rückruf + Schritt)	Payloads, die bei Schrittversuchen des Absenders zurückgegeben wurden, plus zwei Kopien der Callback-Nutzlast

Zusammengesetzte Operationen

Zusammengesetzte Operationen kombinieren mehrere dauerhafte Operationen, um komplexe Koordinationsmuster wie parallele Ausführung, Array-Verarbeitung und verschachtelte Kontexte zu verarbeiten:

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Daten wurden beibehalten
Parallel	Gestartet	1 + N Zweige (1 übergeordneter Kontext + N untergeordnete Kontexte)	Bis zu zwei Kopien der zurückgegebenen Nutzdatengröße aus jedem Zweig plus die Status der einzelnen Zweige
Zuordnung	Gestartet	Über 1 N Zweige (1 übergeordneter Kontext + N untergeordnete Kontexte)	Bis zu zwei Kopien der zurückgegebenen Nutzdatengröße aus jeder Iteration plus die Status jeder Iteration
Promise Helferlein	Completed	1	Hat die Payload-Größe aus dem Versprechen zurückgegeben

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Daten wurden beibehalten
RunInChildContext	Succeeded/Failed	1	Die Nutzdatengröße wurde aus dem untergeordneten Kontext zurückgegeben

Für Kontexte wie von `runInChildContext` Ergebnisse, die kleiner als 256 KB sind, werden direkt mit Checkpoints verglichen oder intern für zusammengesetzte Operationen verwendet. Größere Ergebnisse werden nicht gespeichert, sondern während der Wiedergabe rekonstruiert, indem die Operationen des Kontexts erneut verarbeitet werden.

Unterstützte Laufzeiten für langlebige Funktionen

Für ausgewählte verwaltete Laufzeiten und OCI-Container-Images stehen langlebige Funktionen zur Verfügung, die zusätzliche Flexibilität bei der Laufzeitversion bieten. Sie können robuste Funktionen für Python Node.js, Java und C# (.NET) mithilfe verwalteter Laufzeiten direkt in der Konsole oder programmgesteuert über Infrastructure-as-Code erstellen.

Von Lambda verwaltete Laufzeiten

Die folgenden verwalteten Laufzeiten unterstützen dauerhafte Funktionen, wenn Sie Funktionen in der Lambda-Konsole erstellen oder den AWS CLI Parameter with verwenden. `--durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'` Vollständige Informationen zu Lambda-Laufzeiten finden Sie unter Lambda-Laufzeiten. ???

Sprache	Laufzeit
Node.js	nodejs22.x
Node.js	nodejs24.x
Python	Python 3.13
Python	Python 3.14

Sprache	Laufzeit
Java	java17
Java	java 21
Java	Java 25
C# (.NET)	dotnet 8
C# (.NET)	dotnet10

> **Note**
>
> Lambda Node.js - und Python-Laufzeiten enthalten das robuste Ausführungs-SDK zum Testen und Entwickeln. Wir empfehlen jedoch, das SDK in Ihr Bereitstellungspaket für die Produktion aufzunehmen. Dies gewährleistet die Versionskonsistenz und vermeidet potenzielle Laufzeitaktualisierungen, die sich auf Ihr Funktionsverhalten auswirken könnten. Da Java und C# (.NET) kompilierte Sprachen sind, enthalten Lambda Java- und .NET-Runtimes das Durable Execution SDK nicht, weshalb es in Ihrem Bereitstellungspaket enthalten sein muss.

Node.js

Installieren Sie das SDK in Ihrem Projekt: Node.js

```
npm install @aws/durable-execution-sdk-js
```

Das SDK unterstützt JavaScript und TypeScript. Für TypeScript Projekte enthält das SDK Typdefinitionen.

Python

Installieren Sie das SDK in Ihrem Python-Projekt:

```
pip install aws-durable-execution-sdk-python
```

Das Python-SDK verwendet synchrone Methoden und erfordert `async/await` keine.

Java

Fügen Sie eine Abhängigkeit hinzu zu`pom.xml`:

```
<dependency>
    <groupId>software.amazon.lambda.durable</groupId>
    <artifactId>aws-durable-execution-sdk-java</artifactId>
    <version>VERSION</version>
</dependency>
```

Installieren Sie das SDK in Ihrem Java-Projekt:

```
mvn install
```

Das Java SDK bietet sowohl synchrone als auch asynchrone Versionen jeder Methode.

C# (.NET)

Fügen Sie das SDK zu Ihrem .NET-Projekt hinzu:

```
dotnet add package Amazon.Lambda.DurableExecution
```

Das .NET-SDK ermöglicht dauerhafte Operationen über`IDurableContext`. Ihr Handler erhält den Durable Execution Service Envelope und delegiert an. `DurableFunction.WrapAsync` Jeder Operationskörper benötigt eine`CancellationToken`.

Container-Images

Sie können langlebige Funktionen mit Container-Images verwenden, um zusätzliche Laufzeitversionen oder benutzerdefinierte Laufzeitkonfigurationen zu unterstützen. Mit Container-Images können Sie Laufzeitversionen verwenden, die nicht als verwaltete Laufzeiten verfügbar sind, oder Ihre Laufzeitumgebung anpassen.

So erstellen Sie eine dauerhafte Funktion mithilfe eines Container-Images:

1. Erstellen Sie ein Dockerfile, das auf einem Lambda-Basisimage basiert
2. Installieren Sie das Durable Execution SDK in Ihrem Container
3. Erstellen Sie das Container-Image und übertragen Sie es in Amazon Elastic Container Registry
4. Erstellen Sie die Lambda-Funktion aus dem Container-Image, wobei die dauerhafte Ausführung aktiviert ist

Beispiel für einen Container

Erstellen Sie ein Dockerfile:

Python

Erstellen Sie ein Dockerfile für Python 3.11:

```
FROM public.ecr.aws/lambda/python:3.11

# Copy requirements file
COPY requirements.txt ${LAMBDA_TASK_ROOT}/

# Install dependencies including durable SDK
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}/

# Set the handler
CMD [ "lambda_function.handler" ]
```

Erstelle eine Datei: `requirements.txt`

```
aws-durable-execution-sdk-python
```

Java

Erstellen Sie ein Dockerfile für Java 25:

```
FROM --platform=linux/amd64 public.ecr.aws/lambda/java:25

# Install Maven
RUN dnf install -y maven

WORKDIR /var/task

# Copy Maven configuration and source code
COPY pom.xml .
COPY src ./src

# Build
RUN mvn clean package -DskipTests

# Move JAR to lib directory
RUN mv target/*.jar lib/

# Set the handler
CMD ["src.path.to.lambdaFunction::handler"]
```

Erstellen und pushen Sie das Image:

```
# Build the image
docker build -t my-durable-function .

# Tag for ECR
docker tag my-durable-function:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-
durable-function:latest

# Push to ECR
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-function:latest
```

Erstellen Sie die Funktion mit aktivierter dauerhafter Ausführung:

```
aws lambda create-function \
  --function-name myDurableFunction \
```

```
  --package-type Image \
  --code ImageUri=123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-
function:latest \
  --role arn:aws:iam::123456789012:role/lambda-execution-role \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'
```

Weitere Informationen zur Verwendung von Container-Images mit Lambda finden Sie unter Erstellen von Lambda-Container-Images im Lambda Developer Guide.

Überlegungen zur Laufzeit

SDK-Versionsverwaltung: Um die volle Kontrolle über Ihre Abhängigkeiten zu behalten und mögliche Probleme mit der Versionsfehlausrichtung zu vermeiden, empfehlen wir Ihnen, alle Abhängigkeiten Ihrer Funktion zu Ihrem Bereitstellungspaket hinzuzufügen, auch wenn Versionen davon standardmäßig in der Lambda-Laufzeit enthalten sind. Dazu gehört auch das Durable Execution SDK. Sperren Sie das Durable Execution SDK auf eine Hauptversion in Ihrer Abhängigkeitsdatei. Mit einer neuen Hauptversion können Änderungen eingeführt werden, die zu Fehlschlägen bei der Ausführung während des Fluges führen können. Verwenden Sie nummerierte Versionen oder Aliase für langlebige Funktionen, anstatt `$LATEST` sicherzustellen, dass Änderungen der SDK-Version die Ausführung während des Fluges nicht beeinträchtigen.

AWS Runtime-Updates: aktualisiert verwaltete Laufzeiten um Sicherheitspatches und Bugfixes. Diese Updates können neue SDK-Versionen enthalten. Um unerwartetes Verhalten zu vermeiden, nehmen Sie das SDK in Ihr Bereitstellungspaket auf und testen Sie es gründlich, bevor Sie es für die Produktion bereitstellen.

Größe des Container-Images: Container-Images haben eine unkomprimierte Maximalgröße von 10 GB. Das Durable Execution SDK fügt Ihrem Image eine minimale Größe hinzu. Optimieren Sie Ihren Container, indem Sie mehrstufige Builds verwenden und unnötige Abhängigkeiten entfernen.

Kaltstartleistung: Container-Images haben möglicherweise längere Kaltstartzeiten als verwaltete Laufzeiten. Das Durable Execution SDK hat nur minimale Auswirkungen auf die Kaltstartleistung. Verwenden Sie Provisioned Concurrency, wenn die Kaltstart-Latenz für Ihre Anwendung entscheidend ist.

Dauerhafte Lambda-Funktionen aufrufen

Dauerhafte Lambda-Funktionen unterstützen dieselben Aufrufmethoden wie andere Lambda-Funktionen. Sie können langlebige Funktionen synchron, asynchron oder über Zuordnungen von

Ereignisquellen aufrufen. Der Aufrufvorgang ist mit anderen Funktionen identisch, langlebige Funktionen bieten jedoch zusätzliche Funktionen für Ausführungen mit langer Laufzeit und automatische Statusverwaltung.

Methoden zum Aufrufen

Synchroner Aufruf: Rufen Sie eine dauerhafte Funktion auf und warten Sie auf die Antwort. Synchrone Aufrufe sind auf 15 Minuten begrenzt, oder weniger, wenn das Funktions- oder Ausführungstimeout niedriger ist. Verwenden Sie synchrone Aufrufe, wenn Sie sofortige Ergebnisse benötigen oder wenn Sie APIs und Dienste integrieren, die eine Antwort erwarten. Sie können Wait-Operationen für effiziente Berechnungen verwenden, ohne den Aufrufer zu unterbrechen — der Aufruf wartet, bis die gesamte Durable-Ausführung abgeschlossen ist. Verwenden Sie für idempotente Ausführungsstarts den Parameter für den Ausführungsnamen, wie unter Idempotenz beschrieben.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Asynchroner Aufruf: Ein Ereignis zur Verarbeitung in eine Warteschlange stellen, ohne auf eine Antwort zu warten. Lambda stellt das Ereignis in eine Warteschlange und kehrt sofort zurück. Asynchrone Aufrufe unterstützen Ausführungsdauern von bis zu einem Jahr. Verwenden Sie asynchrone Aufrufe für Fire-and-Forget-Szenarien oder wenn die Verarbeitung im Hintergrund erfolgen kann. Verwenden Sie für idempotente Ausführungsstarts den Parameter für den Ausführungsnamen, wie unter Idempotenz beschrieben.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Zuordnungen von Ereignisquellen: Konfigurieren Sie Lambda so, dass Ihre dauerhafte Funktion automatisch aufgerufen wird, wenn Datensätze von stream- oder warteschlangenbasierten Diensten wie Amazon SQS, Kinesis oder DynamoDB verfügbar sind. Zuordnungen von Ereignisquellen fragen die Ereignisquelle ab und rufen Ihre Funktion mit Datensatzstapeln auf. Einzelheiten zur Verwendung von Ereignisquellenzuordnungen mit dauerhaften Funktionen, einschließlich Beschränkungen für die Ausführungsdauer, finden Sie unter Zuordnungen von Ereignisquellen mit dauerhaften Funktionen.

Vollständige Informationen zu den einzelnen Aufrufmethoden finden Sie unter synchroner Aufruf und asynchroner Aufruf.

> **Note**
>
> Dauerhafte Funktionen unterstützen Dead-Letter-Queues (DLQs) zur Fehlerbehandlung, unterstützen jedoch keine Lambda-Destinationen. Konfigurieren Sie einen DLQ, um Datensätze von fehlgeschlagenen Aufrufen zu erfassen.

Qualifizierte ARNs erforderlich

Dauerhafte Funktionen erfordern qualifizierte Bezeichner für den Aufruf. Sie müssen dauerhafte Funktionen mit einer Versionsnummer, einem Alias oder aufrufen. `$LATEST` Sie können entweder einen vollqualifizierten ARN oder einen Funktionsnamen mit version/alias Suffix verwenden. Sie können keinen unqualifizierten Bezeichner (ohne Versions- oder Aliassuffix) verwenden.

Gültige Aufrufe:

```
# Using full ARN with version number
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1

# Using full ARN with alias
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:prod

# Using full ARN with $LATEST
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:$LATEST

# Using function name with version number
my-durable-function:1

# Using function name with alias
my-durable-function:prod
```

Ungültige Aufrufe:

```
# Unqualified ARN (not allowed)
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function

# Unqualified function name (not allowed)
my-durable-function
```

Diese Anforderung stellt sicher, dass langlebige Ausführungen während ihres gesamten Lebenszyklus konsistent bleiben. Wenn eine dauerhafte Ausführung gestartet wird, wird sie an die Funktionsversion angeheftet, mit der sie gestartet wurde. Wenn Ihre Funktion Stunden oder Tage später angehalten und wieder aufgenommen wird, ruft Lambda dieselbe Version auf, sodass sich die Veröffentlichung einer neuen Version nur auf neue Ausführungen auswirkt. In ähnlicher Weise verwenden neue Ausführungen, wenn Sie einen Alias aktualisieren, die neue Version, und laufende Ausführungen werden mit der Version fortgesetzt, mit der sie gestartet wurden.

⚠ **`$LATEST`-Ausführungen können bei aktualisiertem** Code wieder aufgenommen werden

Ausführungen, die mit dem `$LATEST` Qualifier gestartet wurden, sind nicht an eine feste Kopie Ihres Codes angeheftet. Wenn Sie Ihren Funktionscode aktualisieren, während eine Ausführung unterbrochen ist, wird die Ausführung mit dem neuen Code fortgesetzt und nicht mit der Version, mit der sie gestartet wurde.
Ihr Code verarbeitet den gespeicherten Ausführungsstatus möglicherweise nicht mehr auf dieselbe Weise, was zu Fehlern führt, die nicht deterministisch sind oder zu unbemerkten Fehlern führen. Weitere Hinweise zum Schreiben von Funktionen, die sicher wiedergegeben werden können, finden Sie unter. Funktionsversionen und Aliase

Bewährte Methode

Verwenden Sie nummerierte Versionen oder Aliase eher für langlebige Produktionsfunktionen als. `$LATEST` Nummerierte Versionen sind unveränderlich und unterstützen die deterministische Wiedergabe. Optional bieten Aliase eine stabile Referenz, die Sie aktualisieren können, um auf neue Versionen zu verweisen, ohne den Aufrufcode zu ändern. Verwenden Sie diese `$LATEST` Option nur

für die Prototypenerstellung oder zur Verkürzung der Bereitstellungszeiten während der Entwicklung, da Ausführungen möglicherweise nicht korrekt wiedergegeben werden, wenn sich der Code während der Ausführung ändert.

Den Ausführungslebenszyklus verstehen

Wenn Sie eine dauerhafte Funktion aufrufen, erstellt Lambda eine dauerhafte Ausführung, die sich über mehrere Funktionsaufrufe erstrecken kann:

1. Erster Aufruf: Ihre Aufrufanforderung erstellt eine neue dauerhafte Ausführung. Lambda weist eine eindeutige Ausführungs-ID zu und beginnt mit der Verarbeitung.
2. Ausführung und Checkpointing: Während Ihre Funktion dauerhafte Operationen ausführt, erstellt das SDK Checkpoints, die den Fortschritt verfolgen.
3. Aussetzung (falls erforderlich): Wenn Ihre Funktion dauerhafte Wartezeiten wie `wait` oder oder oder `waitForCallback` automatische Schrittwiederholungen verwendet, unterbricht Lambda die Ausführung und beendet die Berechnung der Rechenzeit.
4. Wiederaufnahme: Wenn es Zeit ist, fortzufahren (auch nach erneuten Versuchen), ruft Lambda Ihre Funktion erneut auf. Das SDK spielt das Checkpoint-Protokoll erneut ab und setzt dort fort, wo die Ausführung unterbrochen wurde.
5. Abschluss: Wenn Ihre Funktion ein Endergebnis zurückgibt oder einen unbehandelten Fehler ausgibt, wird die dauerhafte Ausführung abgeschlossen.

Bei synchronen Aufrufen wartet der Aufrufer, bis die gesamte dauerhafte Ausführung abgeschlossen ist, einschließlich aller Warteoperationen. Wenn die Ausführung das Aufruf-Zeitlimit (15 Minuten oder weniger) überschreitet, wird das Timeout für den Aufruf überschritten. Bei asynchronen Aufrufen kehrt Lambda sofort zurück und die Ausführung wird unabhängig fortgesetzt. Verwenden Sie die dauerhaften Ausführungs-APIs, um den Ausführungsstatus zu verfolgen und die Endergebnisse abzurufen.

Aus dem Anwendungscode aufrufen

Verwenden Sie die AWS SDKs, um dauerhafte Funktionen aus Ihrem Anwendungscode aufzurufen, und zwar mit derselben `Invoke` API, die Sie für jede Lambda-Funktion verwenden. Fügen Sie eine Versionsnummer, einen Alias oder `$LATEST` in den Funktionsnamen ein:

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';

const client = new LambdaClient({});

// Synchronous invocation
const response = await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  Payload: JSON.stringify({ orderId: '12345' })
}));

const result = JSON.parse(Buffer.from(response.Payload!).toString());

// Asynchronous invocation
await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  InvocationType: 'Event',
  Payload: JSON.stringify({ orderId: '12345' })
}));
```

Python

```
import boto3
import json

client = boto3.client('lambda')

# Synchronous invocation
response = client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    Payload=json.dumps({'orderId': '12345'})
)

result = json.loads(response['Payload'].read())

# Asynchronous invocation
client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
```

```
    InvocationType='Event',
    Payload=json.dumps({'orderId': '12345'})
)
```

Verkettete Aufrufe

Dauerhafte Funktionen können jede Lambda-Funktion aufrufen, einschließlich einer anderen dauerhaften Funktion, indem sie die `invoke` Operation von verwenden. `DurableContext` Dadurch wird ein verketteter Aufruf erzeugt, bei dem die aufrufende Funktion angehalten wird, bis die aufgerufene Funktion abgeschlossen ist:

TypeScript

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Invoke another durable function and wait for result
    const result = await context.invoke(
      'process-order',
      'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
      { orderId: event.orderId }
    );

    return { statusCode: 200, body: JSON.stringify(result) };
  }
);
```

Python

```
@durable_execution
def handler(event, context: DurableContext):
    # Invoke another durable function and wait for result
    result = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
        {'orderId': event['orderId']},
        name='process-order'
    )
```

```
    return {'statusCode': 200, 'body': json.dumps(result)}
```

Verkettete Aufrufe erzeugen einen Checkpoint in der aufrufenden Funktion. Wenn die aufrufende Funktion unterbrochen wird, wird sie vom Checkpoint aus mit dem Ergebnis der aufgerufenen Funktion fortgesetzt, ohne die Funktion erneut aufzurufen.

> **Note**
>
> Cross-account Verkettete Aufrufe werden nicht unterstützt. Die aufgerufene Funktion muss sich in demselben AWS Konto wie die aufrufende Funktion befinden.

Zuordnungen von Ereignisquellen mit dauerhaften Funktionen

Dauerhafte Funktionen funktionieren mit allen Lambda-Ereignisquellenzuordnungen. Konfigurieren Sie Ereignisquellenzuordnungen für dauerhafte Funktionen auf die gleiche Weise, wie Sie sie für andere Funktionen konfigurieren. Ereignisquellenzuordnungen fragen automatisch Ereignisquellen wie Amazon SQS, Kinesis und DynamoDB Streams ab und rufen Ihre Funktion mit Datensatzstapeln auf.

Zuordnungen von Ereignisquellen sind nützlich für langlebige Funktionen, die Streams oder Warteschlangen mit komplexen, mehrstufigen Workflows verarbeiten. Sie können beispielsweise eine dauerhafte Funktion erstellen, die Amazon SQS SQS-Nachrichten mit Wiederholungen, externen API-Aufrufen und menschlichen Genehmigungen verarbeitet.

Wie Zuordnungen von Ereignisquellen dauerhafte Funktionen aufrufen

Bei Zuordnungen von Ereignisquellen werden dauerhafte Funktionen synchron aufgerufen. Sie warten, bis die vollständige Ausführung abgeschlossen ist, bevor der nächste Stapel verarbeitet oder Datensätze als verarbeitet markiert werden. Wenn die gesamte dauerhafte Ausführungszeit das geltende Timeout-Limit überschreitet (standardmäßig 15 Minuten oder bis zu 90 Minuten für Funktionen, die auf Lambda Managed Instances ausgeführt werden), wird die Ausführung unterbrochen und schlägt fehl. Die Ereignisquellenzuordnung empfängt eine Timeout-Ausnahme und behandelt sie entsprechend ihrer Wiederholungskonfiguration.

Begrenzung der Ausführungsdauer

Wenn dauerhafte Funktionen durch Zuordnungen von Ereignisquellen aufgerufen werden, darf die Gesamtdauer der dauerhaften Ausführung das maximale Funktions-Timeout nicht überschreiten: 15 Minuten für Funktionen, die im Standardkapazitätsmodus (auf Abruf) ausgeführt werden, oder bis zu 90 Minuten für Funktionen, die auf Lambda Managed Instances ausgeführt werden (außer Amazon MQ- und Amazon DocumentDB DocumentDB-Ereignisquellenzuordnungen, die auf 15 Minuten begrenzt bleiben). Dieses Limit gilt für die gesamte dauerhafte Ausführung von Anfang bis Ende, nicht nur für einzelne Funktionsaufrufe.

Das Funktions-Timeout begrenzt jeden Aufruf. Das dauerhafte Ausführungstimeout begrenzt die Gesamtzeit vom Start bis zum Abschluss der Ausführung. Bei Zuordnungen von Ereignisquellen haben beide dasselbe Maximum: standardmäßig 15 Minuten oder 90 Minuten auf Lambda Managed Instances.

Beispielszenarien:

- Gültig (beliebiger Kapazitätsmodus): Eine dauerhafte Funktion verarbeitet eine Amazon SQS SQS-Nachricht in drei Schritten, die jeweils 2 Minuten dauern, und wartet dann 5 Minuten, bevor ein letzter Schritt abgeschlossen wird. Gesamtausführungszeit: 11 Minuten. Dies funktioniert in beiden Kapazitätsmodi, da es unter dem Standardlimit von 15 Minuten liegt.
- Nur für Lambda Managed Instances gültig: Eine dauerhafte Funktion verarbeitet eine Amazon SQS SQS-Nachricht, schließt die erste Verarbeitung in 2 Minuten ab und wartet dann 20 Minuten auf einen externen Rückruf, bevor der Vorgang abgeschlossen wird. Gesamtausführungszeit: 22 Minuten. Dies überschreitet das Standardlimit von 15 Minuten, liegt jedoch innerhalb des 90-Minuten-Limits für Funktionen, die auf Lambda Managed Instances ausgeführt werden.
- Ungültig (beliebiger Kapazitätsmodus): Eine dauerhafte Funktion verarbeitet einen Kinesis-Datensatz mit mehreren Wartevorgängen, die zwischen den Schritten insgesamt zwei Stunden dauern. Dies überschreitet sogar das 90-Minuten-Limit für Lambda Managed Instances, sodass die Ausführung unabhängig vom Kapazitätsmodus fehlschlägt. Verwenden Sie das zwischengeschaltete Funktionsmuster für Workflows mit einer solchen Länge.

> **Important**
>
> Konfigurieren Sie Ihr dauerhaftes Ausführungstimeout innerhalb des geltenden Grenzwerts, wenn Sie Ereignisquellenzuordnungen verwenden, da andernfalls die Erstellung der Ereignisquellenzuordnung fehlschlägt. Das Limit beträgt standardmäßig 15 Minuten oder bis zu 90 Minuten bei Lambda Managed Instances. Wenn Ihr

Workflow längere Ausführungszeiten erfordert, verwenden Sie das unten beschriebene Zwischenfunktionsmuster.

Zuordnungen von Ereignisquellen konfigurieren

Konfigurieren Sie mithilfe der Lambda-Konsole oder SDKs Ereignisquellenzuordnungen für dauerhafte Funktionen. AWS CLI AWS Alle Eigenschaften der Ereignisquellenzuordnung, wie Batchgröße und Batching-Fenster, gelten für dauerhafte Funktionen:

```
aws lambda create-event-source-mapping \
  --function-name arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1
 \
  --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
  --batch-size 10 \
  --maximum-batching-window-in-seconds 5
```

Verwenden Sie einen qualifizierten ARN mit einer Versionsnummer, einem Alias oder `$LATEST` wenn Sie eine Ereignisquellenzuordnung für eine dauerhafte Funktion konfigurieren.

Fehlerbehandlung bei Zuordnungen von Ereignisquellen

Zuordnungen von Ereignisquellen bieten eine integrierte Fehlerbehandlung, die mit dauerhaften Funktionen funktioniert:

- Wiederholungsverhalten: Wenn der erste Aufruf fehlschlägt, versucht die Zuordnung der Ereignisquelle entsprechend ihrer Wiederholungskonfiguration erneut. Konfigurieren Sie die maximale Anzahl an Wiederholungsversuchen und die Wiederholungsintervalle entsprechend Ihren Anforderungen.
- Dead-letter Warteschlangen: Konfigurieren Sie eine Warteschlange mit unerlaubten Briefen, um Datensätze aufzuzeichnen, die nach allen Wiederholungen fehlschlagen. Dies verhindert den Verlust von Nachrichten und ermöglicht die manuelle Überprüfung fehlgeschlagener Datensätze.
- Teilweise Batchfehler: Verwenden Sie für Amazon SQS und Kinesis die Meldung teilweiser Batch-Fehler, um Datensätze einzeln zu verarbeiten und nur fehlgeschlagene Datensätze erneut zu versuchen.
- Bei Fehler halbieren: Aktivieren Sie für Kinesis- und DynamoDB Streams die Option Halbieren bei Fehler, um fehlgeschlagene Batches aufzuteilen und problematische Datensätze zu isolieren.

> **Note**
>
> Dauerhafte Funktionen unterstützen Dead-Letter-Queues (DLQs) zur Fehlerbehandlung, unterstützen jedoch keine Lambda-Destinationen. Konfigurieren Sie einen DLQ, um Datensätze von fehlgeschlagenen Aufrufen zu erfassen.

Vollständige Informationen zur Fehlerbehandlung bei der Zuordnung von Ereignisquellen finden Sie unter Zuordnungen von Ereignisquellen.

Verwendung einer Zwischenfunktion für Workflows mit langer Laufzeit

Wenn Ihr Workflow mehr als das geltende Limit für die Zuordnung von Ereignisquellen erfordert (standardmäßig 15 Minuten oder 90 Minuten bei Lambda Managed Instances), verwenden Sie eine zwischengeschaltete Standard-Lambda-Funktion zwischen der Zuordnung der Ereignisquelle und Ihrer dauerhaften Funktion. Die Zwischenfunktion empfängt Ereignisse aus der Ereignisquellenzuordnung und ruft die dauerhafte Funktion asynchron auf, wodurch die Beschränkung der Ausführungsdauer aufgehoben wird.

Dieses Muster entkoppelt das synchrone Aufrufmodell der Ereignisquellenzuordnung vom Modell der dauerhaften Funktion mit langer Laufzeit. Bei der Zuordnung der Ereignisquellen wird die Zwischenfunktion aufgerufen, die nach dem Start der dauerhaften Ausführung schnell zurückkehrt. Die dauerhafte Funktion wird dann so lange wie nötig unabhängig ausgeführt (bis zu 1 Jahr).

Architektur

Das Muster der zwischengeschalteten Funktionen besteht aus drei Komponenten:

1. Zuordnung der Ereignisquelle: Ruft die Ereignisquelle ab (Amazon SQS, Kinesis, DynamoDB Streams) und ruft die Zwischenfunktion synchron mit Datensatzstapeln auf.
2. Zwischenfunktion: Eine Lambda-Funktion ohne dauerhafte Ausführung, die Ereignisse aus der Ereignisquellenzuordnung empfängt, die Daten bei Bedarf validiert und transformiert und die dauerhafte Funktion asynchron aufruft. Diese Funktion wird schnell abgeschlossen (normalerweise in weniger als 1 Sekunde) und gibt die Steuerung an die Ereignisquellenzuordnung zurück.
3. Dauerhafte Funktion: Verarbeitet das Ereignis mit einer komplexen, mehrstufigen Logik, die über längere Zeiträume ausgeführt werden kann. Wird asynchron aufgerufen, sodass es nicht durch die Begrenzung der Ausführungsdauer der Ereignisquellenzuordnung eingeschränkt ist.

Implementierung

Die Zwischenfunktion empfängt das gesamte Ereignis aus der Ereignisquellenzuordnung und ruft die dauerhafte Funktion asynchron auf. Verwenden Sie den Parameter für den Ausführungsnamen, um sicherzustellen, dass die idempotente Ausführung gestartet wird, und verhindern Sie so eine doppelte Verarbeitung, falls die Ereignisquellenzuordnung erneut versucht:

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';
import { SQSEvent } from 'aws-lambda';
import { createHash } from 'crypto';

const lambda = new LambdaClient({});

export const handler = async (event: SQSEvent) => {
  // Invoke durable function asynchronously with execution name
  await lambda.send(new InvokeCommand({
    FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType: 'Event',
    Payload: JSON.stringify({
      executionName: event.Name,
      event: event
    })
  }));

  return { statusCode: 200 };
};
```

Python

```
import boto3
import json
import hashlib

lambda_client = boto3.client('lambda')

def handler(event, context):
    # Invoke durable function asynchronously with execution name
```

```
    lambda_client.invoke(
        FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
        InvocationType='Event',
        Payload=json.dumps({
            'executionName': execution_name,
            'event': event["name"]
        })
    )

    return {'statusCode': 200}
```

Verwenden Sie Powertools für Idempotenz in der Zwischenfunktion selbst, um doppelte Aufrufe der dauerhaften Funktion AWS Lambda zu verhindern, falls die Zuordnung der Ereignisquelle die Zwischenfunktion erneut versucht.

Die dauerhafte Funktion empfängt die Nutzdaten mit dem Namen der Ausführung und verarbeitet alle Datensätze mit lang andauernder Logik:

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (payload: any, context: DurableContext) => {
    const sqsEvent = payload.event;

    // Process each record with complex, multi-step logic
    const results = await context.map(
      sqsEvent.Records,
      async (ctx, record) => {
        const validated = await ctx.step('validate', async () => {
          return validateOrder(JSON.parse(record.body));
        });

        // Wait for external approval (could take hours or days)
        const approval = await ctx.waitForCallback(
          'approval',
          async (callbackId) => {
```

```
            await requestApproval(callbackId, validated);
          },
          { timeout: { hours: 48 } }
        );

        // Complete processing
        return await ctx.step('complete', async () => {
          return completeOrder(validated, approval);
        });
      }
    );

    return { statusCode: 200, processed: results.getResults().length };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
from aws_durable_execution_sdk_python.config import Duration, WaitForCallbackConfig
from collections.abc import Sequence
import json

def validate_order(order_data: dict) -> dict:
    """Validate order data - always passes."""
    return order_data

def request_approval(callback_id: str, validated_order: dict) -> None:
    """Request approval for the order - always passes."""
    pass

def complete_order(validated_order: dict, approval_result: str) -> dict:
    """Complete the order processing - always passes."""
    return validated_order

@durable_execution
def lambda_handler(payload, context: DurableContext):
    sqs_event = payload['event']

    def process_record(
        ctx: DurableContext,
```

```
        record: dict,
        index: int,
        items: Sequence[dict]
    ) -> dict:
        validated = ctx.step(
            lambda _: validate_order(json.loads(record['body'])),
            name=f'validate-{index}'
        )

        approval = ctx.wait_for_callback(
            submitter=lambda callback_id, wait_ctx: request_approval(callback_id,
validated),
            name=f'approval-{index}',
            config=WaitForCallbackConfig(timeout=Duration.from_seconds(172800))
        )

        return ctx.step(
            lambda _: complete_order(validated, approval),
            name=f'complete-{index}'
        )

    results = context.map(
        inputs=sqs_event['Records'],
        func=process_record,
        name='process-records'
    )

    return {
        'statusCode': 200,
        'started': results.started_count,
        'completed': results.success_count,
        'failed': results.failure_count,
        'total': results.total_count
    }
```

Wesentliche Überlegungen

Durch dieses Muster wird die Beschränkung der Ausführungsdauer bei der Zuordnung der Ereignisquelle aufgehoben, indem die Zuordnung der Ereignisquelle von der dauerhaften Ausführung entkoppelt wird. Die Zwischenfunktion kehrt unmittelbar nach dem Start der dauerhaften Ausführung

zurück, sodass die Verarbeitung durch die Zuordnung der Ereignisquelle fortgesetzt werden kann. Die dauerhafte Funktion wird dann so lange wie nötig unabhängig ausgeführt.

Die Zwischenfunktion ist erfolgreich, wenn sie die dauerhafte Funktion aufruft, nicht wenn die dauerhafte Ausführung abgeschlossen ist. Wenn die dauerhafte Ausführung später fehlschlägt, versucht die Zuordnung der Ereignisquelle nicht erneut, da der Stapel bereits erfolgreich verarbeitet wurde. Implementieren Sie die Fehlerbehandlung in der dauerhaften Funktion und konfigurieren Sie Warteschlangen für fehlerhafte Ausführungen.

Verwenden Sie den Parameter für den Ausführungsnamen, um sicherzustellen, dass die idempotente Ausführung gestartet wird. Wenn die Zuordnung der Ereignisquelle erneut versucht, die Zwischenfunktion zu verwenden, startet die dauerhafte Funktion keine doppelte Ausführung, da der Ausführungsname bereits vorhanden ist.

Unterstützte Ereignisquellen

Dauerhafte Funktionen unterstützen alle Lambda-Ereignisquellen, die Ereignisquellenzuordnungen verwenden:

- Amazon SQS SQS-Warteschlangen (Standard und FIFO)
- Kinesis-Streams
- DynamoDB Streams
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon MQ (ActiveMQ und RabbitMQ)
- Amazon DocumentDB DocumentDB-Change-Streams

Alle Arten von Ereignisquellen unterliegen dem zuvor beschriebenen dauerhaften Ausführungslimit (standardmäßig 15 Minuten), wenn dauerhafte Funktionen aufgerufen werden. Bei Lambda Managed Instances erhöht sich das Limit für diese Ereignisquellen auf 90 Minuten, mit Ausnahme von Amazon MQ und Amazon DocumentDB, die weiterhin auf 15 Minuten begrenzt sind.

Wiederholungen für langlebige Lambda-Funktionen

Dauerhafte Funktionen bieten automatische Wiederholungsfunktionen, die Ihre Anwendungen widerstandsfähig gegen vorübergehende Ausfälle machen. Das SDK verarbeitet Wiederholungen auf zwei Ebenen: schrittweise Wiederholungen bei Ausfällen der Geschäftslogik und Backend-Wiederholungen bei Infrastrukturausfällen.

Schrittweise Wiederholungen

Wenn innerhalb eines Schritts eine nicht abgefangene Ausnahme auftritt, wiederholt das SDK den Schritt automatisch auf der Grundlage der konfigurierten Wiederholungsstrategie. Wiederholungen von Schritten sind Operationen, bei denen das SDK die Ausführung unterbrechen und später fortsetzen kann, ohne dass der Fortschritt verloren geht.

Verhalten bei schrittweisen Wiederholungen

In der folgenden Tabelle wird beschrieben, wie das SDK Ausnahmen in Schritten behandelt:

Szenario	Was passiert	Messung der Auswirkungen
Ausnahme im Gleichschritt mit den verbleibenden Wiederholungsversuchen	Das SDK erstellt einen Checkpoint für den erneuten Versuch und unterbricht die Funktion. Beim nächsten Aufruf wiederholt sich der Schritt mit der konfigurierten Backoff-Verzögerung.	1 Vorgang + Fehler, Payload-Größe
Ausnahme im Schritt ohne verbleibende Wiederholungsversuche	Der Schritt schlägt fehl und löst eine Ausnahme aus. Wenn Ihr Handler-Code diese Ausnahme nicht catch, schlägt die gesamte Ausführung fehl.	1 Vorgang + Fehler, Größe der Nutzlast

Wenn ein Schritt erneut versucht werden muss, überprüft das SDK den Wiederholungsstatus und beendet den Lambda-Aufruf, wenn keine andere Arbeit ausgeführt wird. Auf diese Weise kann das SDK Backoff-Verzögerungen implementieren, ohne Rechenressourcen zu verbrauchen. Die Funktion wird nach Ablauf der Backoff-Periode automatisch wieder aufgenommen.

Konfiguration von Strategien zur schrittweisen Wiederholung

Konfigurieren Sie Wiederholungsstrategien, um zu steuern, wie Schritte mit Fehlern umgehen. Sie können maximale Versuche, Backoff-Intervalle und Bedingungen für Wiederholungsversuche angeben. Eine vollständige Referenz zu Hilfsprogrammen, Voreinstellungen und benutzerdefinierten Strategien für Wiederholungsstrategien finden Sie in der Dokumentation zum Durable Execution SDK unter Wiederholungen.

Ausnahmen außerhalb von Schritten

Wenn in Ihrem Handlercode, aber außerhalb eines Schritts eine nicht abgefangene Ausnahme auftritt, markiert das SDK die Ausführung als fehlgeschlagen. Dadurch wird sichergestellt, dass Fehler in Ihrer Anwendungslogik ordnungsgemäß erfasst und gemeldet werden.

Szenario	Was passiert	Messung der Auswirkungen
Ausnahme im Handler-Code außerhalb eines beliebigen Schritts	Das SDK markiert die Ausführung als FEHLGESCHLAGEN und gibt den Fehler zurück. Die Ausnahme wird nicht automatisch wiederholt.	Fehler bei der Größe der Payload

Um die automatische Wiederholung von fehleranfälligem Code zu aktivieren, schließen Sie den Vorgang in einem Schritt mit einer Wiederholungsstrategie ab. Schritte bieten automatische Wiederholungsversuche mit konfigurierbarem Backoff, während Code außerhalb von Schritten sofort fehlschlägt.

Der Aufruf wird erneut versucht

Wiederholungen auf Aufrufebene werden unterschiedlich behandelt, je nachdem, wie versucht wird, die dauerhafte Lambda-Funktion aufzurufen. In der folgenden Tabelle wird beschrieben, wie sich die verschiedenen Aufruftypen auf die Wiederholungsversuche auf Aufrufebene auswirken können.

Aufruftyp	Was passiert
Synchroner Aufruf	Lambda wiederholt den Aufruf bei einem Fehler bei der Ausführung dauerhafter Funktionen nicht automatisch. Wiederholungsversuche bei fehlgeschlagenen Aufrufen hängen von der Quelle des synchronen Aufrufs ab. Beispielsweise mithilfe des AWS SDK. Standardmäßig ThrottlingException werden sie InternalFailure automatisch wiederholt.
Asynchroner Aufruf	Wenn die Ausführung einer dauerhaften Funktion fehlschlägt (z. B. wenn sie in den Status FAILED,

Aufruftyp	Was passiert
	STOPPED oder TIMED_OUT übergeht), wiederholt Lambda die Ausführung nicht erneut. Dies unterscheidet sich von Funktionen ohne dauerhafte Ausführung, bei denen Lambda die Funktion erneut versucht, wenn asynchrone Aufrufe fehlschlagen. Die `MaximumRetryAttempts` Einstellung für asynchrone Aufrufe gilt nicht für dauerhafte Ausführungen. Wenn Sie eine Dead-Letter-Warteschlange (DLQ) für die Funktion konfigurieren, sendet Lambda das auslösende Ereignis an die DLQ.
ESM (Zuordnung der Ereignisquellen)	Lambda versucht standardmäßig den gesamten Stapel erneut, bis er erfolgreich ist. Für Stream-Quellen (DynamoDB und Kinesis) können Sie die maximale Anzahl von Wiederholungsversuchen von Lambda konfigurieren, wenn Ihre Funktion einen Fehler zurückgibt. Siehe Batching von Ereignisquellenzuordnungen. Für Amazon SQS ESM können Sie maximale Wiederholungsversuche über eine DLQ in der ursprünglichen Amazon SQS SQS-Warteschlange konfigurieren. Siehe Amazon SQS ESM konfigurieren. Als bewährte Methode könnten Sie einen DLQ auf Funktionsebene in Betracht ziehen und Lambda sendet das fehlgeschlagene auslösende Ereignis an den DLQ. Siehe Funktion DLQ.

Aufruftyp	Was passiert
Direkter Trigger	Das hängt vom „Trigger“ ab. Lambda verarbeitet beispielsweise Funktionen, die durch Amazon S3 S3-Ereignisbenachrichtigungen ausgelöst werden, asynchron. Weitere Informationen finden Sie unter Verarbeiten von Amazon SQS SQS-Ereignisbenachrichtigungen mit Lambda. Lambda verarbeitet Funktionen, die durch Amazon SNS SNS-Ereignisbenachrichtigungen ausgelöst werden, asynchron. Weitere Informationen finden Sie unter Aufrufen von Lambda-Funktionen mit Amazon SNS SNS-Benachrichtigungen. Das Verhalten beim erneuten Versuch eines asynchronen Aufrufs ist weiter oben im Tabelleneintrag „Asynchroner Aufruf“ beschrieben. Wenn Amazon SNS Lambda nicht erreichen kann, oder die Nachricht abgelehnt wird, wiederholt Amazon SNS den Vorgang in zunehmenden Intervallen über mehrere Stunden. Weitere Details finden Sie unter Zuverlässigkeit in den häufig gestellten Fragen zu Amazon SNS. API Gateway ruft Lambda synchron auf und gibt die echte Fehlerantwort an den Anforderer zurück. Siehe Wiederholungsversuche bei Aufrufen. Das Verhalten bei wiederholten Aufrufen ist weiter oben im Tabelleneintrag „Synchroner Aufruf“ beschrieben. Weitere Informationen finden Sie unter den einzelnen direkten Triggern.

Wiederholungen im Backend

Backend-Wiederholungen treten auf, wenn Lambda auf Infrastrukturausfälle oder Laufzeitfehler stößt oder wenn das SDK nicht mit dem Durable Execution Service kommunizieren kann. Lambda versucht diese Fehler automatisch erneut, um Ihre dauerhaften Funktionen bei der Wiederherstellung vorübergehender Infrastrukturprobleme zu unterstützen.

Szenarien für Wiederholungsversuche im Backend

Lambda versucht automatisch, Ihre Funktion erneut auszuführen, wenn sie auf die folgenden Szenarien trifft:

- Interne Dienstfehler — Wenn Lambda oder der Durable Execution Service einen 5xx-Fehler zurückgibt, was auf ein vorübergehendes Serviceproblem hinweist.
- Drosselung — Wenn Ihre Funktion aufgrund von Parallelitätsbeschränkungen oder Servicekontingenten gedrosselt wird.
- Timeouts — Wenn das SDK den Durable Execution Service innerhalb des Timeout-Zeitraums nicht erreichen kann.
- Sandbox-Initialisierungsfehler — Wenn Lambda die Ausführungsumgebung nicht initialisieren kann.
- Laufzeitfehler — Wenn die Lambda-Laufzeit auf Fehler außerhalb Ihres Funktionscodes stößt, wie z. B. Speicherfehler oder Prozessabstürze.
- Ungültige Checkpoint-Token-Fehler — Wenn das Checkpoint-Token nicht mehr gültig ist, was in der Regel auf dienstseitige Statusänderungen zurückzuführen ist.

In der folgenden Tabelle wird beschrieben, wie das SDK mit diesen Szenarien umgeht:

Szenario	Was passiert	Messung der Auswirkungen
Laufzeitfehler außerhalb des Durable-Handlers (OOM, Timeout, Absturz)	Lambda versucht den Aufruf automatisch erneut. Das SDK wiederholt die Wiedergabe ab dem letzten Checkpoint, wobei abgeschlossene Schritte übersprungen werden.	Fehler: Payload-Größe + 1 Vorgang pro Wiederholung
Dienstfehler (5xx) oder Timeout beim Aufrufen von//APIs `CheckpointDurableExecution` `GetDurableExecutionState`	Lambda versucht den Aufruf automatisch erneut. Das SDK wiederholt die Wiedergabe ab dem letzten Checkpoint.	Fehler: Payload-Größe + 1 Vorgang pro Wiederholungsversuch
Drosselung (429) oder ungültiges Checkpoint-Token beim Aufrufen von//APIs `CheckpointDurableExecution`	Lambda wiederholt den Aufruf automatisch mit exponentiellem Backoff. Das SDK wiederholt die Wiedergabe ab dem letzten Checkpoint.	Fehler: Payload-Größe + 1 Vorgang pro Wiederholungsversuch

Szenario	Was passiert	Messung der Auswirkungen
`GetDurableExecutionState`		
Client-Fehler (4xx, außer 429 und ungültigem Token) bei/APIs `CheckpointDurableExecution GetDurableExecutionState`	Das SDK markiert die Ausführung als FEHLGESCHLAGEN. Es erfolgt kein automatischer Wiederholungsversuch, da der Fehler auf ein permanentes Problem hinweist.	Fehler bei der Größe der Payload

Backend-Wiederholungen verwenden exponentielles Backoff und werden fortgesetzt, bis die Funktion erfolgreich ist oder das Ausführungstimeout erreicht ist. Während der Wiedergabe überspringt das SDK abgeschlossene Prüfpunkte und setzt die Ausführung ab dem letzten erfolgreichen Vorgang fort, um sicherzustellen, dass Ihre Funktion abgeschlossene Arbeit nicht erneut ausführt.

Versuchen Sie es erneut mit bewährten Methoden

Beachten Sie bei der Konfiguration von Wiederholungsstrategien die folgenden bewährten Methoden:

- Konfigurieren Sie explizite Wiederholungsstrategien — Verlassen Sie sich nicht auf das standardmäßige Wiederholungsverhalten in der Produktion. Konfigurieren Sie Strategien für explizite Wiederholungen mit einer für Ihren Anwendungsfall angemessenen maximalen Anzahl von Versuchen und Backoff-Intervallen.
- Verwenden Sie bedingte Wiederholungen — Implementieren Sie `shouldRetry` Logik, um nur vorübergehende Fehler (Ratenbegrenzungen, Timeouts) zu wiederholen und bei dauerhaften Fehlern (Validierungsfehler, nicht gefunden) schnell fehlzuschlagen.
- Legen Sie eine angemessene maximale Anzahl von Versuchen fest — ein Gleichgewicht zwischen Belastbarkeit und Ausführungszeit. Zu viele Wiederholungen können die Fehlererkennung verzögern, während zu wenige Versuche zu unnötigen Fehlern führen können.
- Exponentielles Backoff verwenden — Exponentielles Backoff reduziert die Belastung nachgeschalteter Dienste und erhöht die Wahrscheinlichkeit einer Wiederherstellung nach vorübergehenden Ausfällen.

- Fehleranfälligen Code schrittweise zusammenfassen — Code außerhalb von Schritten kann nicht automatisch wiederholt werden. Integrieren Sie externe API-Aufrufe, Datenbankabfragen und andere fehleranfällige Operationen schrittweise mit Wiederholungsstrategien.
- Überwachen Sie Wiederholungsmetriken — Verfolgen Sie schrittweise Wiederholungsvorgänge und Ausführungsfehler in Amazon CloudWatch , um Muster zu identifizieren und Wiederholungsstrategien zu optimieren.

Idempotenz

Dauerhafte Funktionen bieten eine eingebaute Idempotenz für Ausführungsstarts anhand von Ausführungsnamen. Wenn Sie einen Ausführungsnamen angeben, verwendet Lambda ihn, um doppelte Ausführungen zu verhindern und sichere Wiederholungsversuche von Aufrufanforderungen zu ermöglichen. Schritte haben standardmäßig eine Semantik für die Ausführung mindestens einmal. Während der Wiedergabe gibt das SDK Checkpoint-Ergebnisse zurück, ohne dass abgeschlossene Schritte erneut ausgeführt werden. Ihre Geschäftslogik muss jedoch idempotent sein, um potenzielle Wiederholungen vor Abschluss zu bewältigen.

> **Note**
>
> Lambda-Ereignisquellenzuordnungen (ESM) unterstützen beim Start keine Idempotenz. Daher startet jeder Aufruf (einschließlich Wiederholungen) eine neue dauerhafte Ausführung. Um eine idempotente Ausführung mit Zuordnungen von Ereignisquellen sicherzustellen, implementieren Sie entweder eine Idempotenzlogik in Ihren Funktionscode, z. B. mit Powertools for, AWS Lambda oder verwenden Sie eine reguläre Lambda-Funktion als Proxy (Dispatcher), um eine dauerhafte Funktion mit einem Idempotenzschlüssel (Ausführungsnamenparameter) aufzurufen.

Namen der Ausführung

Sie können einen Ausführungsnamen angeben, wenn Sie eine dauerhafte Funktion aufrufen. Der Ausführungsname fungiert als Idempotenzschlüssel, sodass Sie Aufrufanforderungen sicher wiederholen können, ohne dass es zu doppelten Ausführungen kommt. Wenn Sie keinen Namen angeben, generiert Lambda automatisch eine eindeutige Ausführungs-ID.

Die Namen der Ausführung müssen innerhalb Ihres Kontos und Ihrer Region eindeutig sein. Wenn Sie eine Funktion mit einem bereits vorhandenen Ausführungsnamen aufrufen, hängt das Lambda-Verhalten vom Status der vorhandenen Ausführung ab und davon, ob die Nutzlast übereinstimmt.

Verhalten bei Idempotenz

In der folgenden Tabelle wird beschrieben, wie Lambda Aufrufanforderungen behandelt, je nachdem, ob Sie einen Ausführungsnamen angeben, welchen Ausführungsstatus Sie haben und ob die Nutzlast übereinstimmt:

Szenario	Name angegeben?	Bestehender Ausführungsstatus	Ist die Nutzlast identisch?	Behavior
1	Nein	N/A	N/A	Neue Ausführung gestartet: Lambda generiert eine eindeutige Ausführungs-ID und startet eine neue Ausführung
2	Ja	Existierte nie oder die Aufbewahrung ist abgelaufen	N/A	Neue Ausführung gestartet: Lambda startet eine neue Ausführung mit dem angegebenen Namen
3	Ja	In Ausführung	Ja	Idempotenter Start: Lambda gibt die vorhandenen Ausführungsinformationen zurück, ohne ein Duplikat zu starten. Bei synchronen Aufrufen dient dies als erneutes Anhängen an die laufende Ausführung
4	Ja	In Ausführung	Nein	Fehler: Lambda gibt einen `DurableExecutionAl`

Szenario	Name angegeben?	Bestehender Ausführungsstatus	Ist die Nutzlast identisch?	Behavior
				`readyStartedException` Fehler zurück, da eine Ausführung mit diesem Namen bereits mit einer anderen Nutzlast ausgeführt wird
5	Ja	Geschlossen (erfolgreich, fehlgeschlagen, gestoppt oder Zeitlimit überschritten)	Ja	Idempotenter Start: Lambda gibt die vorhandenen Ausführungsinformationen zurück, ohne eine neue Ausführung zu starten. Das Ergebnis der geschlossenen Ausführung wird zurückgegeben
6	Ja	Geschlossen (erfolgreich, fehlgeschlagen, gestoppt oder Zeitlimit überschritten)	Nein	Fehler: Lambda gibt einen `DurableExecutionAlreadyStartedException` Fehler zurück, da eine Ausführung mit diesem Namen bereits mit einer anderen Nutzlast abgeschlossen wurde

Hinweis

Die Szenarien 3 und 5 zeigen ein idempotentes Verhalten, bei dem Lambda doppelte Aufrufanforderungen sicher behandelt, indem es vorhandene Ausführungsinformationen zurückgibt, anstatt Duplikate zu erstellen.

Schritt: Idempotenz

Schritte haben standardmäßig eine Semantik, die mindestens einmal ausgeführt wird. Wenn Ihre Funktion nach einer Wartezeit, einem Rückruf oder einem Fehler erneut ausgeführt wird, vergleicht das SDK jeden Schritt mit dem Checkpoint-Log. Für Schritte, die bereits abgeschlossen wurden, gibt das SDK das Checkpoint-Ergebnis zurück, ohne die Schrittlogik erneut auszuführen. Wenn jedoch ein Schritt fehlschlägt oder die Funktion unterbrochen wird, bevor der Schritt abgeschlossen ist, kann der Schritt mehrmals ausgeführt werden.

Die Geschäftslogik in Ihren Schritten muss idempotent sein, um potenzielle Wiederholungsversuche zu bewältigen. Verwenden Sie Idempotenz-Schlüssel, um sicherzustellen, dass Operationen wie Zahlungen oder Datenbankschreibvorgänge nur einmal ausgeführt werden, auch wenn der Schritt erneut versucht wird. Einzelheiten zum Programmieren für Idempotenz finden Sie unter Idempotenz und Wiederholungen im AWS Durable Execution SDK Developer Guide.

Testen langlebiger Lambda-Funktionen

Verwenden Sie die SDKs für Durable Execution-Tests, um Ausführungen sowohl lokal als auch in der Cloud auszuführen und zu überprüfen. Autorentests, Assertionen, den Cloud Runner, die SAM-CLI-Integration und vollständige Beispiele finden Sie unter Testen im AWS Durable Execution SDK Developer Guide.

IAM-Berechtigungen für Cloud-Tests

Wenn Sie den Cloud Runner verwenden oder `sam remote invoke` eine bereitgestellte dauerhafte Funktion testen, benötigt der aufrufende Principal die Erlaubnis, die Funktion aufzurufen und ihren Ausführungsverlauf zu lesen. Ordnen Sie Ihrem Testaufrufer die folgenden Berechtigungen zu:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:GetDurableExecution",
                "lambda:GetDurableExecutionHistory"
            ],
            "Resource": [
                "arn:aws:lambda:region:account-id:function:function-name",
                "arn:aws:lambda:region:account-id:function:function-name:*"
            ]
        }
    ]
}
```

Ersetzen Sie *regionaccount-id*, und *function-name* durch Ihre Werte.

Fehler beim Debuggen

Wenn Tests fehlschlagen, überprüfen Sie das Ausführungsergebnis, um zu verstehen, was schief gelaufen ist. Überprüfen Sie den Ausführungsstatus, um festzustellen, ob die Funktion erfolgreich war, fehlgeschlagen ist oder ob das Zeitlimit überschritten wurde. Lesen Sie die Fehlermeldungen, um die Fehlerursache zu verstehen.

Untersuchen Sie die einzelnen Betriebsergebnisse, um herauszufinden, wo das Verhalten von den Erwartungen abwich. Überprüfen Sie die Ergebnisse der einzelnen Schritte, um zu sehen, welche Werte erzielt wurden. Überprüfen Sie die Reihenfolge der Operationen, um zu bestätigen, dass die Operationen in der erwarteten Reihenfolge ausgeführt wurden. Zählen Sie die Operationen, um sicherzustellen, dass die richtige Anzahl von Schritten, Wartezeiten und Callbacks erstellt wurde.

Zu den häufigsten Problemen gehören nicht deterministischer Code, der bei der Wiedergabe zu unterschiedlichen Ergebnissen führt, gemeinsamer Status durch globale Variablen, der während der Wiedergabe unterbrochen wird, und fehlende Operationen aufgrund bedingter Logikfehler. Verwenden Sie Standard-Debugger und Protokollierung, um den Funktionscode Schritt für Schritt durchzugehen und den Ausführungsablauf zu verfolgen.

Sehen Sie sich bei Cloud-Tests den Ausführungsverlauf in den CloudWatch Protokollen an, um detaillierte Betriebsprotokolle einzusehen. Verwenden Sie Tracing, um den Ausführungsablauf zwischen Diensten zu verfolgen und Engpässe zu identifizieren.

Überwachung langlebiger Funktionen

Sie können Ihre dauerhaften Funktionen überwachen mit CloudWatch Metriken, CloudWatch Protokolle und Ablaufverfolgung. Da langlebige Funktionen über einen längeren Zeitraum ausgeführt werden können und sich über mehrere Funktionsaufrufe erstrecken, erfordert ihre Überwachung das Verständnis ihrer jeweiligen Ausführungsmuster, einschließlich Checkpoints, Zustandsübergängen und Wiedergabeverhalten.

> **Note**
>
> Weitere Informationen zu den Kontingenten, die für langlebige Funktionen gelten, finden Sie unter Kontingente für dauerhafte Funktionen.

CloudWatch Metriken

Lambda veröffentlicht automatisch Metriken in CloudWatch ohne zusätzliche Kosten. Stabile Funktionen bieten zusätzliche Metriken, die über die standardmäßigen Lambda-Metriken hinausgehen und Ihnen helfen, Workflows mit langer Laufzeit, Statusmanagement und Ressourcenauslastung zu überwachen.

Dauerhafte Ausführungsmetriken

Lambda gibt die folgenden Metriken für dauerhafte Ausführungen aus:

Metrik	Description
`ApproximateRunningDurableExecutions`	Anzahl der dauerhaften Ausführungen im Status RUNNING
`ApproximateRunningDurableExecutionsUtilization`	Prozentsatz der aktuell in Anspruch genommenen Höchstzahl an laufenden Dauerausführungen Ihres Kontos
`DurableExecutionDuration`	Verstrichene Wanduhrzeit in Millisekunden, während der eine dauerhafte Ausführung im Status RUNNING verharrte
`DurableExecutionStarted`	Anzahl der gestarteten dauerhaften Ausführungen

Metrik	Description
`DurableExecutionStopped`	Anzahl der dauerhaften Ausführungen, die mit dem Befehl beendet wurden StopDurableExecution API
`DurableExecutionSucceeded`	Anzahl der dauerhaften Ausführungen, die erfolgreich abgeschlossen wurden
`DurableExecutionFailed`	Anzahl der dauerhaften Ausführungen, die mit einem Fehler abgeschlossen wurden
`DurableExecutionTimedOut`	Anzahl der dauerhaften Ausführungen, bei denen das konfigurierte Ausführungs-Timeout überschritten wurde
`DurableExecutionOperations`	Kumulierte Anzahl der Operationen, die im Rahmen einer dauerhaften Ausführung ausgeführt wurden
`DurableExecutionStorageWrittenBytes`	Kumulierte Datenmenge in Byte, die bei einer dauerhaften Ausführung beibehalten wurden

CloudWatch Metriken

Lambda gibt Standardmetriken für Aufruf, Leistung und Parallelität für langlebige Funktionen aus. Da eine dauerhafte Ausführung mehrere Funktionsaufrufe umfassen kann, während sie Checkpoints und Wiederholungen durchläuft, verhalten sich diese Metriken anders als bei Standardfunktionen:

- Aufrufe: Zählt jeden Funktionsaufruf, einschließlich Wiederholungen. Eine einzige dauerhafte Ausführung kann mehrere Aufrufdatenpunkte generieren.
- Dauer: Misst jeden Funktionsaufruf separat. Verwenden `DurableExecutionDuration` für die Gesamtzeit, die eine einzige dauerhafte Ausführung benötigt.
- Fehler: Verfolgt Fehler beim Aufrufen von Funktionen. Verwenden `DurableExecutionFailed` für Fehler auf Ausführungsebene.

Eine vollständige Liste der Standard-Lambda-Metriken finden Sie unter Arten von Metriken für Lambda-Funktionen.

Erstellen CloudWatch Alarme

Erstellen CloudWatch Alarme, um Sie zu benachrichtigen, wenn Metriken Schwellenwerte überschreiten. Zu den häufigsten Alarmen gehören:

- `ApproximateRunningDurableExecutionsUtilization` überschreitet 80% Ihrer Quote
- `DurableExecutionFailed` steigt über einen Schwellenwert
- `DurableExecutionTimedOut` zeigt an, dass das Zeitlimit für Ausführungen überschritten wird
- `DurableExecutionStorageWrittenBytes` nähert sich den Speichergrenzen

Weitere Informationen finden Sie unter Verwenden CloudWatch Alarme.

EventBridge Veranstaltungen

Lambda veröffentlicht dauerhafte Ereignisse zur Änderung des Ausführungsstatus an EventBridge. Sie können diese Ereignisse verwenden, um Workflows auszulösen, Benachrichtigungen zu senden oder Änderungen im Ausführungszyklus Ihrer dauerhaften Funktionen nachzuverfolgen.

Dauerhafte Ereignisse zur Änderung des Ausführungsstatus

Lambda sendet ein Ereignis an EventBridge immer dann, wenn eine dauerhafte Ausführung ihren Status ändert. Diese Ereignisse haben die folgenden Merkmale:

- Quelle: `aws.lambda`
- Typ des Details: `Durable Execution Status Change`

Statusänderungsereignisse werden für die folgenden Ausführungszustände veröffentlicht:

- `RUNNING` — Die Ausführung wurde gestartet
- `SUCCEEDED` - Die Ausführung wurde erfolgreich abgeschlossen
- `STOPPED` - Die Ausführung wurde mit dem beendet StopDurableExecution API
- `FAILED` - Die Ausführung ist mit einem Fehler fehlgeschlagen
- `TIMED_OUT` - Die Ausführung hat das konfigurierte Timeout überschritten

Das folgende Beispiel zeigt ein dauerhaftes Ereignis zur Änderung des Ausführungsstatus:

```
{
  "version": "0",
  "id": "d019b03c-a8a3-9d58-85de-241e96206538",
  "detail-type": "Durable Execution Status Change",
  "source": "aws.lambda",
  "account": "123456789012",
  "time": "2025-11-20T13:08:22Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "durableExecutionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST/durable-execution/090c4189-b18b-4296-9d0c-cfd01dc3a122/9f7d84c9-ea3d-3ffc-
b3e5-5ec51c34ffc9",
    "durableExecutionName": "order-123",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:2",
    "status": "RUNNING",
    "startTimestamp": "2025-11-20T13:08:22.345Z"
  }
}
```

Für Terminalzustände (`SUCCEEDED`, `STOPPED`, `FAILED`, `TIMED_OUT`), beinhaltet die Veranstaltung eine `endTimestamp` Feld, das angibt, wann die Ausführung abgeschlossen ist.

Erstellen EventBridge Regeln

Erstellen EventBridge Regeln, um Ereignisse mit dauerhaften Änderungen des Ausführungsstatus an Ziele wie Amazon Simple Notification Service, Amazon Simple Queue Service oder andere Lambda-Funktionen weiterzuleiten.

Im folgenden Beispiel wird eine Regel erstellt, die allen dauerhaften Änderungen des Ausführungsstatus entspricht:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"]
}
```

Im folgenden Beispiel wird eine Regel erstellt, die nur auf fehlgeschlagene Ausführungen zutrifft:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "status": ["FAILED"]
  }
}
```

Im folgenden Beispiel wird eine Regel erstellt, die Statusänderungen für eine bestimmte Funktion abgleicht:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "functionArn": [{
      "prefix": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
    }]
  }
}
```

Weitere Informationen zum Erstellen EventBridge Regeln finden Sie unter Amazon EventBridge Anleitungen in der EventBridge Benutzerleitfaden.

Verwenden von Warteschlangen mit unzustellbaren Briefen EventBridge Veranstaltungen

EventBridge Statusänderungsereignisse informieren in Echtzeit, wenn eine dauerhafte Ausführung in den Status FAILED, STOPPED oder TIMED_OUT übergeht. Sie enthalten jedoch nicht die ursprüngliche Nutzlast des Ereignisses, das die Ausführung ausgelöst hat. Um das auslösende Ereignis für eine spätere Analyse oder erneute Verarbeitung zu erfassen, konfigurieren Sie eine Dead-Letter-Queue (DLQ) in Ihrer dauerhaften Funktion.

Wenn eine dauerhafte Ausführung nach einem asynchronen Aufruf fehlschlägt, sendet Lambda das ursprüngliche auslösende Ereignis an den konfigurierten DLQ. Verwenden EventBridge Ereignisse und DLQs zusammen für eine umfassende Strategie zur Fehlerbehandlung:

- EventBridge Veranstaltungen — Löse sofortige Warnmeldungen aus, aktualisiere Dashboards oder starte Workflows zur Problembehebung, wenn die Ausführung fehlschlägt.

- Dead-letter Warteschlangen — Behalten Sie die ursprüngliche Nutzlast für Ereignisse bei, sodass Sie untersuchen können, was den Fehler verursacht hat, das Problem debuggen und das Ereignis optional erneut verarbeiten können, nachdem Sie das zugrunde liegende Problem behoben haben.

Informationen zur Konfiguration eines DLQ für Ihre Funktion finden Sie unter Dead-letter Warteschlangen.

AWS X-Ray Nachverfolgung

Sie können aktivieren X-Ray Rückverfolgung Ihrer dauerhaften Funktionen. Lambda besteht die X-Ray Trace-Header zur dauerhaften Ausführung, sodass Sie Anfragen in Ihrem gesamten Workflow verfolgen können.

Um zu aktivieren X-Ray; Tracing mit der Lambda-Konsole, wählen Sie Ihre Funktion aus, wählen Sie dann Tools für Konfiguration, Überwachung und Betrieb aus und aktivieren Sie Active Tracing unter X-Ray.

Um zu aktivieren X-Ray Ablaufverfolgung mit dem AWS CLI:

```
aws lambda update-function-configuration \
    --function-name my-durable-function \
    --tracing-config Mode=Active
```

Um zu aktivieren AWS X-Ray Ablaufverfolgung mit AWS SAM:

```
Resources:
  MyDurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      DurableConfig:
        ExecutionTimeout: 3600
```

Für weitere Informationen über X-Ray, siehe AWS X-Ray Leitfaden für Entwickler.

Bewährte Methoden für langlebige Lambda-Funktionen

Dauerhafte Funktionen verwenden ein wiedergabebasiertes Ausführungsmodell, das andere Programmierpraktiken erfordert als herkömmliche Lambda-Funktionen. Anleitungen zum Schreiben und Testen von AWS dauerhaftem Workflow-Code finden Sie unter Bewährte Methoden im Durable Execution SDK Developer Guide.

Die folgenden Empfehlungen sind bewährte Methoden für die Bereitstellung, den Aufruf und die Überwachung von dauerhaften Lambda-Funktionen.

Funktionsversionen und Aliase

Rufen Sie Funktionen mit Versionsnummern oder Aliasnamen auf, um Ausführungen an bestimmte Codeversionen zu binden. Stellen Sie sicher, dass neue Codeversionen den Status älterer Versionen verarbeiten können. Benennen Sie Schritte nicht um und ändern Sie ihr Verhalten nicht so, dass die Wiedergabe unterbrochen wird. Wenn Sie einen Schritt umbenennen oder sein Verhalten ändern, während Ausführungen ausgeführt werden, können diese Ausführungen möglicherweise nicht fortgesetzt werden oder zu falschen Ergebnissen führen. Das passiert, weil die Laufzeitumgebung den gespeicherten Status nicht mehr mit der erwarteten Schrittdefinition abgleichen kann.

Überwachen

Aktivieren Sie die strukturierte Protokollierung mit Ausführungs-IDs und Schrittnamen. Richten Sie CloudWatch Alarme für Fehlerraten und Ausführungsdauer ein. Verwenden Sie Tracing, um Engpässe zu identifizieren. Eine ausführliche Anleitung finden Sie unter Überwachung und Debuggen.

Fehlerbehandlung

Zusätzlich zur Konfiguration von Wiederholungsstrategien für vorübergehende Fehler können Sie auf Ihrer dauerhaften Funktion auch eine Warteschlange (Dead-Letter Queue, DLQ) einrichten, um Ereignisse aus dauerhaft fehlgeschlagenen Ausführungen aufzuzeichnen. Wenn eine dauerhafte Ausführung nach einem asynchronen Aufruf einen Terminalstatus (FAILED, STOPPED oder TIMED_OUT) erreicht, sendet Lambda das ursprüngliche auslösende Ereignis an die DLQ. Auf diese Weise können Sie fehlgeschlagene Ereignisse überprüfen, debuggen und optional erneut verarbeiten, ohne sie zu verlieren.

Um eine DLQ zu konfigurieren, setzen Sie die `DeadLetterConfig` Eigenschaft Ihrer Funktion auf eine Amazon SQS SQS-Warteschlange oder einen Amazon SNS SNS-Themen-ARN. Weitere Informationen finden Sie unter Warteschlangen. Dead-letter

Folgen Sie diesen bewährten Methoden zur Fehlerbehandlung mit dauerhaften Funktionen:

- Einen DLQ für asynchrone Aufrufe konfigurieren — Beim asynchronen Aufrufen dauerhafter Funktionen wird immer eine Warteschlange mit unerlaubten Buchstaben angehängt. Im Gegensatz zu asynchronen Aufrufen von Funktionen ohne dauerhafte Ausführung wiederholt Lambda bei einem Fehler nicht automatisch dauerhafte Ausführungen. Eine ordnungsgemäß konfigurierte DLQ erfasst Ereignisse, die zu dauerhaft fehlgeschlagenen Ausführungen geführt haben. Stellen Sie sicher, dass der DLQ über die richtigen Berechtigungen und ausreichend Kapazität für den Empfang von Nachrichten verfügt.
- Schrittweise Wiederholungsstrategien anwenden — Konfigurieren Sie explizite Wiederholungsstrategien mit geeignetem Backoff für vorübergehende Ausfälle. Anleitungen zur Konfiguration von Wiederholungsversuchen finden Sie unter Wiederholungen für dauerhafte Funktionen.
- Kombinieren Sie DLQs mit EventBridge Benachrichtigungen — Verwenden Sie EventBridge Regeln, um bei Statusänderungen FAILED, STOPPED und TIMED_OUT zu warnen, sodass Sie in Echtzeit sehen können, und verwenden Sie eine DLQ, um die ursprüngliche Event-Payload für eine spätere Analyse oder Wiederverarbeitung beizubehalten.
- Überwachen Sie die DLQ-Tiefe — Erstellen Sie für Ihre `ApproximateNumberOfMessagesVisible` DLQ-Metrik einen CloudWatch Alarm, um zu erkennen, wenn sich Fehler häufen.

Zugehörige Ressourcen

- AWS Entwicklerhandbuch für das Durable Execution SDK
- Überwachung langlebiger Lambda-Funktionen
- Wiederholungen für langlebige Lambda-Funktionen
- Testen langlebiger Lambda-Funktionen

Verbesserung der Startleistung mit Lambda SnapStart

Lambda SnapStart kann eine Startleistung von nur unter einer Sekunde bieten, in der Regel ohne Änderungen an Ihrem Event-Funktionscode. SnapStart macht es einfacher, reaktionsschnelle und skalierbare Anwendungen zu erstellen, ohne Ressourcen bereitzustellen oder komplexe Leistungsoptimierungen zu implementieren.

Der größte Beitrag zur Startup-Latenz (oft als Kaltstartzeit bezeichnet) ist die Zeit, die Lambda für die Initialisierung der Funktion aufwendet. Dazu gehören das Laden des Funktionscodes, der Start der Laufzeit und die Initialisierung des Funktionscodes. Mit SnapStart initialisiert Lambda Ihre Event-Funktion, wenn Sie eine Funktionsversion veröffentlichen. Lambda erstellt einen Firecracker MicroVM-Snapshot des Speicher- und Festplattenstatus der initialisierten Ausführungsumgebung, verschlüsselt den Snapshot und speichert ihn intelligent im Cache, um die Abruflatenz zu optimieren.

Um die Stabilität zu gewährleisten, verwaltet Lambda mehrere Kopien jedes Snapshots. Lambda patcht Snapshots und ihre Kopien automatisch mit den neuesten Laufzeit- und Sicherheitsupdates. Wenn Sie die Event Function-Version zum ersten Mal aufrufen und die Aufrufe immer größer werden, nimmt Lambda neue Ausführungsumgebungen aus dem zwischengespeicherten Snapshot wieder auf, anstatt sie von Grund auf neu zu initialisieren, wodurch die Startlatenz verbessert wird.

> **Important**
>
> Wenn Ihre Anwendungen von der Eindeutigkeit des Zustands abhängen, müssen Sie Ihren Funktionscode auswerten und sicherstellen, dass er gegenüber Snapshot-Vorgängen widerstandsfähig ist. Weitere Informationen finden Sie unter Umgang mit Einzigartigkeit mit Lambda SnapStart.

Wenn Sie auf Probleme stoßen, finden Sie weitere Informationen unter. the section called “Fehlerbehebung”

Themen

- Wann verwenden SnapStart
- Unterstützte Funktionen und Einschränkungen
- Unterstützte Regionen
- Erwägungen zur Kompatibilität
- SnapStart Preisgestaltung
- Lambda aktivieren und verwalten SnapStart
- Umgang mit Einzigartigkeit mit Lambda SnapStart
- Implementieren Sie Code vor oder nach Snapshots der Lambda-Ereignisfunktion
- Überwachung für Lambda SnapStart
- Sicherheitsmodell für Lambda SnapStart
- Maximieren Sie die Lambda-Leistung SnapStart

- Behebung SnapStart von Fehlern für Lambda-Ereignisfunktionen

Wann verwenden SnapStart

Lambda SnapStart wurde entwickelt, um die Latenzvariabilität zu beheben, die durch einmaligen Initialisierungscode verursacht wird, z. B. beim Laden von Modulabhängigkeiten oder Frameworks. Beim ersten Aufruf kann es manchmal mehrere Sekunden dauern, bis diese Vorgänge abgeschlossen sind. Wird verwendet SnapStart , um diese Latenz in optimalen Szenarien von mehreren Sekunden auf weniger als eine Sekunde zu reduzieren. SnapStart funktioniert am besten, wenn es mit Funktionsaufrufen in großem Maßstab verwendet wird. Funktionen, die selten aufgerufen werden, weisen möglicherweise nicht dieselben Leistungsverbesserungen auf.

SnapStart ist besonders vorteilhaft für zwei Haupttypen von Anwendungen:

- Latency-sensitive APIs und Benutzerabläufe: Funktionen, die Teil kritischer API-Endpunkte oder benutzerorientierter Abläufe sind, können von SnapStart der geringeren Latenz und den verbesserten Reaktionszeiten profitieren.
- Latency-sensitive Datenverarbeitungs-Workflows: Time-bound Datenverarbeitungs-Workflows, die Lambda Event Functions verwenden, können einen besseren Durchsatz erzielen, indem sie die Latenz bei der Initialisierung von Ausreißerfunktionen reduzieren.

Durch die bereitgestellte Parallelität werden Event Functions initialisiert und sind innerhalb von zweistelligen Millisekunden bereit, darauf zu reagieren. Verwenden Sie die bereitgestellte Parallelität, wenn Ihre Anwendung strenge Kaltstart-Latenzanforderungen hat, die nicht angemessen erfüllt werden können. SnapStart

Unterstützte Funktionen und Einschränkungen

SnapStart unterstützt die folgenden von Lambda verwalteten Laufzeiten und die entsprechenden AWS Lambda-Basisimages im ZIP- und Container-Image-Bereitstellungsformat:

- Java 11 und höher
- Python 3.12 oder höher
- .NET 8 und höher. Wenn Sie das Lambda Annotations Framework für.NET verwenden, führen Sie ein Upgrade auf Amazon.Lambda.AnnotationsVersion 1.6.0 oder höher durch, um die Kompatibilität mit sicherzustellen. SnapStart

Für Container-Images, die auf AWS Lambda-Basis-Images oder benutzerdefinierten Basis-Images basieren, lesen Sie Ihren Funktionscode Implementierung von SnapStart Hooks für Container-Images und überprüfen Sie ihn für die Handhabung von SnapStart Eindeutigkeitsanforderungen.

Andere verwaltete Laufzeiten (wie `nodejs24.x` und`ruby4.0`) OS-only Laufzeiten werden nicht unterstützt. SnapStart unterstützt keine bereitgestellte Parallelität, Amazon Elastic File System (Amazon EFS), Amazon S3 S3-Dateien oder kurzlebigen Speicher mit mehr als 512 MB.

> **Note**
>
> Sie können es SnapStart nur für veröffentlichte Funktionsversionen und Aliase verwenden, die auf Versionen verweisen. Sie können es nicht für die unveröffentlichte Version einer Funktion ($LATEST) verwenden SnapStart .

Unterstützte Regionen

Lambda SnapStart ist in allen Handelsregionen außer Asien-Pazifik (Neuseeland) und Asien-Pazifik (Taipeh) erhältlich.

Erwägungen zur Kompatibilität

Mit SnapStart verwendet Lambda einen einzelnen Snapshot als Ausgangszustand für mehrere Ausführungsumgebungen. Wenn Ihre Funktion während der Initialisierungsphase eines der folgenden Elemente verwendet, müssen Sie vor der Verwendung möglicherweise einige Änderungen vornehmen: SnapStart

Eindeutigkeit

Wenn Ihr Initialisierungscode eindeutigen Inhalt generiert, der im Snapshot enthalten ist, ist der Inhalt möglicherweise nicht eindeutig, wenn er in Ausführungsumgebungen wiederverwendet wird. Um die Einzigartigkeit bei der Verwendung zu wahren SnapStart, müssen Sie nach der Initialisierung eindeutige Inhalte generieren. Dazu gehören eindeutige IDs, eindeutige Geheimnisse und Entropie, die zum Generieren von Pseudozufälligkeiten verwendet wird. Informationen zum Wiederherstellen von Eindeutigkeit finden Sie unter Umgang mit Einzigartigkeit mit Lambda SnapStart.

Netzwerkverbindungen

Der Status der Verbindungen, die Ihre Event-Funktion während der Initialisierungsphase herstellt, ist nicht garantiert, wenn Lambda Ihre Funktion von einem Snapshot aus wieder aufnimmt. Überprüfen Sie den Status Ihrer Netzwerkverbindungen und stellen Sie diese bei Bedarf wieder her. In den meisten Fällen werden Netzwerkverbindungen, die ein AWS SDK herstellt, automatisch wieder aufgenommen. Für andere Verbindungen lesen Sie die bewährten Methoden.

Temporäre Daten

Einige Funktionen laden während der Initialisierungsphase kurzlebige Daten herunter oder initialisieren diese, wie etwa temporäre Anmeldeinformationen oder im Cache gespeicherte Zeitstempel. Aktualisieren Sie kurzlebige Daten im Funktionshandler, bevor Sie sie verwenden, auch wenn Sie sie nicht verwenden. SnapStart

SnapStart Preisgestaltung

> **Note**
>
> Für Java-verwaltete Laufzeiten fallen keine zusätzlichen Kosten für SnapStart an. Die Abrechnung erfolgt basierend auf der Anzahl der Anfragen für Ihre Funktionen, der Zeit, die Ihr Code zum Ausführen benötigt, und dem für Ihre Funktion konfigurierten Arbeitsspeicher.

Die Kosten für die Nutzung SnapStart beinhalten Folgendes:

- Caching: Für jede Funktionsversion, die Sie mit SnapStart aktivierter Option veröffentlichen, zahlen Sie die Kosten für das Zwischenspeichern und Verwalten des Snapshots. Der Preis hängt von der Speichermenge ab, die Sie Ihrer Funktion zuweisen. Ihnen werden mindestens 3 Stunden in Rechnung gestellt.

  Ihnen werden weiterhin Gebühren berechnet, solange Ihre Funktion aktiv ist. Verwenden Sie die ListVersionsByFunctionAPI-Aktion, um Funktionsversionen zu identifizieren, und verwenden Sie sie dann, DeleteFunctionum ungenutzte Versionen zu löschen. Informationen zum automatischen Löschen ungenutzter Funktionsversionen finden Sie im Lambda Version Cleanup-Pattern auf Serverless Land.
- Wiederherstellung: Jedes Mal, wenn eine Funktions-Instance aus einem Snapshot wiederhergestellt wird, zahlen Sie eine Wiederherstellungsgebühr. Der Preis hängt von der Speichermenge ab, die Sie Ihrer Funktion zuweisen.

Wie bei allen Lambda-Ereignisfunktionen fallen Gebühren für die Dauer von Code an, der im Funktionshandler ausgeführt wird. Bei SnapStart Funktionen fallen Dauergebühren auch für Initialisierungscode an, der außerhalb des Handlers deklariert wurde, für die Zeit, die zum Laden der Laufzeit benötigt wird, und für jeden Code, der in einem Runtime-Hook ausgeführt wird. Die Dauer wird vom Beginn der Ausführung Ihres Codes bis zur Rückgabe oder einem anderen Ende berechnet, aufgerundet auf die nächste 1 ms. Lambda verwaltet aus Gründen der Ausfallsicherheit zwischengespeicherte Kopien Ihres Snapshots und wendet automatisch Softwareupdates wie Laufzeit-Upgrades und Sicherheitspatches auf diese an. Jedes Mal, wenn Lambda Ihren Initialisierungscode erneut ausführt, um Softwareupdates zu installieren, fallen Gebühren an.

Weitere Informationen zu den Nutzungskosten finden Sie unter SnapStart AWS Lambda Preisgestaltung.

Lambda aktivieren und verwalten SnapStart

Um sie zu verwenden SnapStart, aktivieren Sie sie SnapStart auf einer neuen oder vorhandenen Lambda-Ereignisfunktion. Veröffentlichen und rufen Sie dann eine Funktionsversion auf.

Themen

- Aktivierung SnapStart (Konsole)
- Aktiviert SnapStart (AWS CLI)
- Aktivierung SnapStart (API)
- Status von Lambda SnapStart - und Eventfunktionen
- Aktualisieren eines Snapshots
- SnapStart Verwenden mit AWS SDKs
- Verwenden mit SnapStart CloudFormation, AWS SAM, und AWS CDK
- Löschen von Snapshots

Aktivierung SnapStart (Konsole)

Zur Aktivierung SnapStart für eine Event-Funktion

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Event-Funktion.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).
4. Wählen Sie im Bereich General configuration (Allgemeine Konfiguration) die Option Edit (Bearbeiten) aus.
5. Wählen Sie auf der Seite Grundeinstellungen bearbeiten für die SnapStartOption Veröffentlichte Versionen aus.
6. Wählen Sie Speichern.
7. Veröffentlichen einer Funktionsversion. Lambda initialisiert Ihren Code, erstellt einen Snapshot der initialisierten Ausführungsumgebung und speichert den Snapshot im Cache für einen Zugriff mit geringer Latenz.
8. Rufen Sie die Funktionsversion auf.

Aktiviert SnapStart (AWS CLI)

Zur Aktivierung SnapStart für eine bestehende Event-Funktion

1. Aktualisieren Sie die Funktionskonfiguration, indem Sie den update-function-configuration-Befehl mit der --snap-start-Option ausführen.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. Veröffentlichen Sie eine Funktionsversion mit dem publish-version-Befehl.

```
aws lambda publish-version \
  --function-name my-function
```

3. Bestätigen Sie, dass sie für die Funktionsversion aktiviert SnapStart ist, indem Sie den Befehl get-function-configuration ausführen und die Versionsnummer angeben. Das folgende Beispiel legt Version 1 fest.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

Wenn in der Antwort angezeigt OptimizationStatuswird, dass dies der 0n Fall ist und der Status lautetActive, dann SnapStart ist diese Option aktiviert und ein Snapshot ist für die angegebene Funktionsversion verfügbar.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. Rufen Sie die Funktionsversion auf, indem Sie den Aufruf-Befehl ausführen und die Version angeben. Das folgende Beispiel ruft Version 1 auf.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
```

```
  response.json
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Zur Aktivierung SnapStart , wenn Sie eine neue Event-Funktion erstellen

1. Erstellen Sie eine Funktion, indem Sie den create-function-Befehl mit der --snap-start-Option ausführen. Geben Sie für --role den Amazon-Ressourcennamen (ARN) Ihrer Ausführungsrolle an.

   > **Note**
   >
   > Wenn Sie eine Container-Image-Funktion verwenden SnapStart , die andere Basis-Images als die AWS Basis-Images von Lambda für Java (Version 11+), Python (Version 3.12+) und .NET (Version 8+) verwendet, folgen Sie den Anweisungen unter, um Ihr Image kompatibel mit Implementierung von SnapStart Hooks für Container-Images zu machen. SnapStart

   ```
   aws lambda create-function \
     --function-name my-function \
     --runtime "java25" \
     --zip-file fileb://my-function.zip \
     --handler my-function.handler \
     --role arn:aws:iam::111122223333:role/lambda-ex \
     --snap-start ApplyOn=PublishedVersions
   ```

2. Erstellen Sie eine Version mit dem publish-version-Befehl.

   ```
   aws lambda publish-version \
     --function-name my-function
   ```

3. Bestätigen Sie, dass diese SnapStart Option für die Funktionsversion aktiviert ist, indem Sie den Befehl get-function-configuration ausführen und die Versionsnummer angeben. Das folgende Beispiel legt Version 1 fest.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

Wenn in der Antwort angezeigt OptimizationStatuswird, dass dies der On Fall ist und der Status lautetActive, dann SnapStart ist diese Option aktiviert und ein Snapshot ist für die angegebene Funktionsversion verfügbar.

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. Rufen Sie die Funktionsversion auf, indem Sie den Aufruf-Befehl ausführen und die Version angeben. Das folgende Beispiel ruft Version 1 auf.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Aktivierung SnapStart (API)

Um zu aktivieren SnapStart

1. Führen Sie eine der folgenden Aktionen aus:
   - Erstellen Sie eine neue Ereignisfunktion mit SnapStart aktiviertem Wert, indem Sie die CreateFunctionAPI-Aktion mit dem SnapStartParameter verwenden.
   - Aktivieren Sie sie SnapStart für eine vorhandene Ereignisfunktion, indem Sie die UpdateFunctionConfigurationAktion mit dem SnapStartParameter verwenden.

2. Veröffentlichen Sie eine Funktionsversion mit der PublishVersionAktion. Lambda initialisiert Ihren Code, erstellt einen Snapshot der initialisierten Ausführungsumgebung und speichert den Snapshot im Cache für einen Zugriff mit geringer Latenz.
3. Bestätigen Sie mithilfe der GetFunctionConfigurationAktion, dass sie für die Funktionsversion aktiviert SnapStart ist. Geben Sie eine Versionsnummer an, um zu bestätigen, dass die Version für diese Version aktiviert SnapStart ist. Wenn in der Antwort angezeigt OptimizationStatuswird, dass dies der Fall ist On und der Status SnapStart lautet`Active`, dann ist aktiviert und ein Snapshot ist für die angegebene Funktionsversion verfügbar.

   ```
   "SnapStart": {
           "ApplyOn": "PublishedVersions",
           "OptimizationStatus": "On"
       },
       "State": "Active",
   ```
4. Rufen Sie die Funktionsversion mit der Aufruf-Aktion auf.

Status von Lambda SnapStart - und Eventfunktionen

Die folgenden Funktionszustände können auftreten, wenn Sie verwenden SnapStart.

Ausstehend

Lambda initialisiert Ihren Code und erstellt einen Snapshot der initialisierten Ausführungsumgebung. Alle Aufrufe oder andere API-Aktionen, die mit der Funktionsversion ausgeführt werden, schlagen fehl.

Aktiv

Die Snapshot-Erstellung ist abgeschlossen und Sie können die Funktion aufrufen. Zur Verwendung SnapStart müssen Sie die veröffentlichte Funktionsversion aufrufen, nicht die unveröffentlichte Version ($LATEST).

Inaktiv

Der Status `Inactive` kann auftreten, wenn Lambda regelmäßig Funktions-Snapshots neu generiert, um Software-Updates anzuwenden. In diesem Fall kann Ihre Funktion in einen `Inactive`-Zustand übergehen, wenn sie nicht initialisiert werden kann.

Bei Funktionen, die eine Java-Laufzeit verwenden, löscht Lambda Snapshots nach 14 Tagen ohne Aufruf. Wenn Sie die Funktionsversion nach 14 Tagen aufrufen, gibt Lambda eine

`SnapStartNotReadyException`-Antwort zurück und beginnt mit der Initialisierung eines neuen Snapshots. Warten Sie, bis die Funktionsversion den `Active`-Status erreicht hat, und rufen Sie es dann erneut auf.

Fehlgeschlagen

Lambda hat beim Ausführen des Initialisierungscodes oder beim Erstellen des Snapshots einen Fehler festgestellt.

Aktualisieren eines Snapshots

Lambda erstellt für jede veröffentlichte Funktionsversion einen Snapshot. Um einen Snapshot zu aktualisieren, veröffentlichen Sie eine neue Funktionsversion.

SnapStart Verwenden mit AWS SDKs

Um AWS SDK-Aufrufe von Ihrer Funktion aus durchzuführen, generiert Lambda einen kurzlebigen Satz von Anmeldeinformationen, indem es die Ausführungsrolle Ihrer Funktion übernimmt. Diese Anmeldeinformationen sind während des Aufrufs Ihrer Funktion als Umgebungsvariablen verfügbar. Sie müssen keine Anmeldeinformationen für das SDK direkt im Code bereitstellen. Standardmäßig überprüft die Kette der Anbieter von Anmeldeinformationen nacheinander jeden Ort, an dem Sie Anmeldeinformationen festlegen können, und wählt die erste verfügbare aus – in der Regel die Umgebungsvariablen (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` und `AWS_SESSION_TOKEN`).

> **Note**
>
> Wenn SnapStart aktiviert, verwendet die Lambda-Laufzeit automatisch die Container-Anmeldeinformationen (`AWS_CONTAINER_CREDENTIALS_FULL_URI`und`AWS_CONTAINER_AUTHORIZATION_TOKEN`) anstelle der Umgebungsvariablen für die Zugriffsschlüssel. Dadurch wird verhindert, dass Anmeldeinformationen ablaufen, bevor die Funktion wiederhergestellt wird.

Verwenden mit SnapStart CloudFormation, AWS SAM, und AWS CDK

- AWS CloudFormation: Deklarieren Sie die SnapStartEntität in Ihrer Vorlage.
- AWS Serverless Application Model (AWS SAM): Deklarieren Sie die SnapStartEigenschaft in Ihrer Vorlage.

- AWS Cloud Development Kit (AWS CDK): Verwenden Sie den SnapStartPropertyTyp.

Löschen von Snapshots

Lambda löscht Snapshots, wenn:

- Sie die Funktion oder Funktionsversion löschen.
- Nur Java-Laufzeiten – Sie die Funktionsversion 14 Tage lang nicht aufrufen. Wenn nach 14 Tagen ohne Aufruf die Funktionsversion in den Status Inaktiv übergeht. Wenn Sie die Funktionsversion nach 14 Tagen aufrufen, gibt Lambda eine `SnapStartNotReadyException`-Antwort zurück und beginnt mit der Initialisierung eines neuen Snapshots. Warten Sie, bis die Funktionsversion den Status Aktiv erreicht hat, und rufen Sie es dann erneut auf.

Lambda entfernt alle Ressourcen, die mit gelöschten Snapshots verknüpft sind, gemäß der Allgemeinen Datenschutzverordnung (DSGVO).

Umgang mit Einzigartigkeit mit Lambda SnapStart

Wenn Aufrufe für eine SnapStart Funktion skaliert werden, verwendet Lambda einen einzigen initialisierten Snapshot, um mehrere Ausführungsumgebungen wieder aufzunehmen. Wenn Ihr Initialisierungscode eindeutigen Inhalt generiert, der im Snapshot enthalten ist, ist der Inhalt möglicherweise nicht eindeutig, wenn er in Ausführungsumgebungen wiederverwendet wird. Um die Einzigartigkeit bei der Verwendung zu wahren SnapStart, müssen Sie nach der Initialisierung eindeutige Inhalte generieren. Dazu gehören eindeutige IDs, eindeutige Geheimnisse und Entropie, die zum Generieren von Pseudozufälligkeiten verwendet wird.

Im Folgenden finden Sie einige bewährte Methoden für die Aufrechterhaltung der Eindeutigkeit Ihres Codes. Für Java-Funktionen bietet Lambda auch ein SnapStart Open-Source-Scan-Tool, mit dessen Hilfe nach Code gesucht werden kann, der Eindeutigkeit voraussetzt. Wenn Sie während der Initialisierungsphase eindeutige Daten generieren, können Sie eine Laufzeit-Hook verwenden, um die Eindeutigkeit wiederherzustellen. Mit Laufzeit-Hooks können Sie bestimmten Code ausführen, unmittelbar bevor Lambda einen Snapshot aufnimmt oder unmittelbar nachdem Lambda eine Funktion aus einem Snapshot fortsetzt.

Vermeiden des Speicherns eines Zustands, der von der Eindeutigkeit während der Initialisierung abhängt

Vermeiden Sie es, während der Initialisierungsphase Ihrer Event-Funktion Daten zwischenzuspeichern, die eindeutig sein sollen, wie z. B. das Generieren einer eindeutigen ID für die Protokollierung oder das Setzen von Ausgangswerten für zufällige Funktionen. Stattdessen empfehlen wir Ihnen, eindeutige Daten zu generieren oder Seeds für Zufallsfunktionen in Ihrem Funktionshandler festzulegen – oder einen Laufzeit-Hook zu verwenden.

Die folgenden Beispiele zeigen, wie eine UUID im Event Function Handler generiert wird.

Java

Example— Generieren einer eindeutigen ID im Event Function Handler

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
```

```
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
    }
  }
```

Python

Example— Generierung einer eindeutigen ID im Event Function Handler

```
import json
import random
import time

unique_number = None

def lambda_handler(event, context):
    seed = int(time.time() * 1000)
    random.seed(seed)
    global unique_number
    if not unique_number:
        unique_number = random.randint(1, 10000)

    print("Unique number: ", unique_number)

    return "Hello, World!"
```

.NET

Example— Generierung einer eindeutigen ID im Event Function Handler

```
namespace Example;
public class SnapstartExample
{
    private Guid _myExecutionEnvironmentGuid;
    public SnapstartExample()
    {
        // This GUID is set for non-restore use cases, such as testing or if
 SnapStart is turned off
        _myExecutionEnvironmentGuid = new Guid();
        // Register the method which will run after each restore. You may need to
 update Amazon.Lambda.Core to see this
```

```
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(MyAfterRestore);
    }

    private ValueTask MyAfterRestore()
    {
        // After restoring this snapshot to a new execution environment, update the
 GUID
        _myExecutionEnvironmentGuid = new Guid();
        return ValueTask.CompletedTask;
    }

    public string Handler()
    {
        return $"Hello World! My Execution Environment GUID is
 {_myExecutionEnvironmentGuid}";
    }
}
```

Kryptografisch sichere Pseudozufallszahlengeneratoren (CSPRNGs) verwenden

Wenn SnapStart aktiviert, erstellt Lambda einen Snapshot der Ausführungsumgebung Ihrer Funktion, einschließlich des gesamten Anwendungs- und Systemspeichers. Das bedeutet, dass der interne Zustand jedes Zufallszahlengenerators (RNG) genau so beibehalten wird, wie er zum Zeitpunkt des Snapshots war, zusammen mit allen zufälligen Bytes, die Ihre Anwendung oder ihre Abhängigkeiten bereits generiert und in Speicherpuffern gespeichert haben.

Wenn Lambda eine Ausführungsumgebung aus einem Snapshot wiederherstellt, wird der Kernel-Zufallszahlengenerator von `/dev/random` und `/dev/urandom` mit neuer Entropie neu geladen. Software, die Zufallszahlen direkt von diesen Geräten liest, behält die Zufälligkeit bei. SnapStart

Wenn Ihre Anwendung von Zufälligkeit abhängt, empfehlen wir Ihnen, kryptographisch sichere Zufallszahlengeneratoren (CSPRNGs) zu verwenden. Zusätzlich zu dem von AL bereitgestellten OpenSSL enthalten die von Lambda verwalteten Laufzeiten, die Unterstützung bieten SnapStart (Java Version 11+, Python Version 3.12+ und .NET Version 8+), die folgenden integrierten CSPRNGs:

- Java: `java.security.SecureRandom`
- Python: `random.SystemRandom`
- .NET: `System.Security.Cryptography.RandomNumberGenerator`

> **Note**
>
> Für die Go Lambda-Laufzeit sind keine Änderungen erforderlich, wenn Ihre Funktionen die Standardbibliothek verwenden`crypto/rand`, die standardmäßig Snapstart-kompatibel ist.

Wenn Sie Ihre Funktion als Container-Image verpacken, bestimmt Ihr Basis-Image die Einzigartigkeit und Kompatibilität mit: SnapStart

- Ein Lambda-Basisimage für eine verwaltete Laufzeit (Java Version 11+, Python Version 3.12+ und .NET Version 8+) — kompatibel mit wie oben beschrieben. SnapStart
- Das bereitgestellte.al2023-Basis-Image — kompatibel, wenn Ihre Programmiersprachen-Runtime Entropie von`/dev/random`, oder dem SnapStart-compatible Build von OpenSSL (openssl-snapsafe-libs) bezieht`/dev/urandom`, der in der Amazon Linux 2023-Paketliste aufgeführt ist. Wenn Ihre Laufzeit auf anderen Entropiequellen basiert, folgen Sie den Schritten, die im Abschnitt „Ihr eigenes Basis-Image“ weiter unten beschrieben sind.
- Ihr eigenes Basis-Image — Prüfen Sie Ihre kryptografischen Bibliotheken und verwenden Sie einen Runtime-Hook nach der Wiederherstellung, um alle zwischengespeicherten Zufallszahlengenerator-Zustände (RNG) oder gepufferte Zufallsbytes zu löschen. So wird sichergestellt, dass die Bibliothek nach der Wiederherstellung neue Entropie aus dem System liest. Rust-Funktionen, die verwenden, müssen beispielsweise nach der Wiederherstellung aufgerufen werden. `rand::rngs::ThreadRng` `reseed()` Wir empfehlen, dass Sie, wann immer möglich, mit einem Lambda-Basisimage erstellen. Insbesondere, wenn Ihr Basis-Image eine eigene Version von OpenSSL oder einen OpenSSL-Fork (wie BoringSSL oder LibreSSL) enthält, wechseln Sie entweder zum mitgelieferten.al2023-Basis-Image, das einen SnapStart-compatible Build von OpenSSL (openssl-snapsafe-libs) enthält, oder verwenden Sie libcrypto (-LC).AWSAWS

Minimale unterstützte Bibliotheksversionen

AWS Kryptografie-Bibliotheken behalten automatisch die Zufälligkeit bei, wobei mit den in der folgenden Tabelle angegebenen Mindestversionen SnapStart begonnen wird. Wenn Sie diese Bibliotheken mit Ihren Lambda-Ereignisfunktionen verwenden, stellen Sie sicher, dass Sie die folgenden Mindestversionen oder spätere Versionen verwenden:

Bibliothek	Unterstützte Mindestversion (x86)	Unterstützte Mindestversion (ARM)
AWS libcrypto (-LC)AWS	1.16.0	1.30.0
AWS libcrypto FIPS	2.0.13	2.0.13

Wenn Sie die vorherigen kryptografischen Bibliotheken mit Ihren Lambda-Ereignisfunktionen als transitive Abhängigkeiten über die folgenden Bibliotheken paketieren, stellen Sie sicher, dass Sie die folgenden Mindestversionen oder spätere Versionen verwenden:

Bibliothek	Unterstützte Mindestversion (x86)	Unterstützte Mindestversion (ARM)
AWS SDK for Java 2.x	2.23.20	2,26,12
AWS Allgemeine Runtime für Java	0.29.8	0,29,25
Kryptoanbieter Amazon Corretto	2.4.1	2.4.1
Amazon Corretto Kryptoanbieter FIPS	2.4.1	2.4.1

Die folgenden Beispiele zeigen, wie CSPRNGs verwendet werden können, um eindeutige Zahlenfolgen zu garantieren, selbst wenn die Funktion aus einem Snapshot wiederhergestellt wird.

Java

Example— java.security. SecureRandom

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
```

```
        System.out.println(rng.next());
      }
      return "Hello, World!";
    }
  }
```

Python

Example— zufällig. SystemRandom

```
import json
import random

secure_rng = random.SystemRandom()

def lambda_handler(event, context):
    random_numbers = [secure_rng.random() for _ in range(10)]

    for number in random_numbers:
        print(number)

    return "Hello, World!"
```

.NET

Example– RandomNumberGenerator

```
using Amazon.Lambda.Core;
using System.Security.Cryptography;
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

namespace DotnetSecureRandom;

public class Function
{
    public string FunctionHandler()
    {
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            byte[] randomUnsignedInteger32Bytes = new byte[4];
            for (int i = 0; i < 10; i++)
            {
```

```
                rng.GetBytes(randomUnsignedInteger32Bytes);
                int randomInt32 = BitConverter.ToInt32(randomUnsignedInteger32Bytes,
 0);
                Console.WriteLine("{0:G}", randomInt32);
            }
        }
        return "Hello World!";
    }
}
```

SnapStart Scan-Tool (nur Java)

Lambda bietet ein Scan-Tool für Java an, um nach Code zu suchen, der Eindeutigkeit voraussetzt. Das SnapStart Scan-Tool ist ein SpotBugsOpen-Source-Plugin, das eine statische Analyse anhand einer Reihe von Regeln durchführt. Das Scan-Tool hilft dabei, potenzielle Codeimplementierungen zu identifizieren, die Annahmen bezüglich der Eindeutigkeit widerlegen könnten. Installationsanweisungen und eine Liste der Prüfungen, die das Scan-Tool durchführt, finden Sie im aws-lambda-snapstart-java-rules-Repository unter. GitHub

Weitere Informationen zum Umgang mit Einzigartigkeit finden Sie unter Schnelleres Starten mit SnapStart im Compute-Blog. AWS Lambda SnapStart AWS

Implementieren Sie Code vor oder nach Snapshots der Lambda-Ereignisfunktion

Sie können Laufzeit-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Funktion aus einem Snapshot fortsetzt. Laufzeit-Hooks sind für eine Vielzahl von Zwecken nützlich, z. B.:

- Bereinigung und Initialisierung: Bevor ein Snapshot erstellt wird, können Sie einen Laufzeit-Hook verwenden, um Bereinigungs- oder Ressourcenfreigabeoperationen durchzuführen. Nachdem ein Snapshot wiederhergestellt wurde, können Sie einen Laufzeit-Hook verwenden, um alle Ressourcen oder Zustände, die nicht im Snapshot erfasst wurden, neu zu initialisieren.
- Dynamische Konfiguration: Sie können Laufzeit-Hooks verwenden, um die Konfiguration oder andere Metadaten dynamisch zu aktualisieren, bevor ein Snapshot erstellt wird oder nachdem er wiederhergestellt wurde. Dies kann nützlich sein, wenn sich Ihre Funktion an Änderungen in der Laufzeitumgebung anpassen muss.
- Externe Integrationen: Sie können Laufzeit-Hooks zur Integration mit externen Diensten oder Systemen verwenden, z. B. zum Senden von Benachrichtigungen oder zum Aktualisieren des externen Status als Teil des Checkpointing- und Wiederherstellungsprozesses.
- Leistungsoptimierung: Sie können Laufzeit-Hooks verwenden, um die Startsequenz Ihrer Funktion fein abzustimmen, z. B. durch Vorladen von Abhängigkeiten. Weitere Informationen finden Sie unter Leistungsoptimierung.

Auf den folgenden Seiten wird erläutert, wie Sie Laufzeit-Hooks für Ihre bevorzugte Laufzeitumgebung implementieren.

Themen

- SnapStart Lambda-Laufzeit-Hooks für Java
- Lambda SnapStart-Laufzeit-Hooks für Python
- SnapStart Lambda-Runtime-Hooks für.NET
- Implementierung von SnapStart Hooks für Container-Images

SnapStart Lambda-Laufzeit-Hooks für Java

Sie können Runtime-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Event-Funktion aus einem Snapshot wieder aufnimmt.

aufzeit-Hooks sind als Teil des Open-Source-Projekts Coordinated Restore at Checkpoint (CRaC) verfügbar. CRaC wird für das Open-Java-Entwicklungs-Kit (OpenJDK) entwickelt. Ein Beispiel für die Verwendung von cRAC mit einer Referenzanwendung finden Sie im cRAC-Repository unter. GitHub CRaC verwendet drei Hauptelemente:

- `Resource` – Eine Schnittstelle mit zwei Methoden, `beforeCheckpoint()` und `afterRestore()`. Verwenden Sie diese Methoden, um den Code zu implementieren, den Sie vor einem Snapshot und nach einer Wiederherstellung ausführen möchten.
- `Context <R extends Resource>` – Um Benachrichtigungen für Prüfpunkte und Wiederherstellungen zu erhalten, muss eine `Resource` mit einem `Context` registriert sein.
- `Core`— Der Koordinationsdienst, der die globale Standardeinstellung `Context` über die statische Methode bereitstellt. `Core.getGlobalContext()`

Weitere Informationen zum `Context` und `Resource` finden Sie unter Paket org.crac in der CRaC-Dokumentation.

Gehen Sie wie folgt vor, um Laufzeit-Hooks mit dem Paket org.crac zu implementieren. Die Lambda-Laufzeit enthält eine angepasste CRaC-Kontextimplementierung, die Ihre Laufzeit-Hooks vor dem Checkpointing und nach der Wiederherstellung aufruft.

Registrierung und Ausführung von Laufzeit-Hooks

Die Reihenfolge, in der Lambda Ihre Laufzeit-Hooks ausführt, wird durch die Reihenfolge der Registrierung bestimmt. Die Reihenfolge der Registrierung entspricht der Reihenfolge des Imports, der Definition oder der Ausführung in Ihrem Code.

- `beforeCheckpoint()`: Wird in umgekehrter Reihenfolge der Registrierung ausgeführt
- `afterRestore()`: Wird in Reihenfolge der Registrierung ausgeführt

Vergewissern Sie sich, dass alle registrierten Hooks ordnungsgemäß importiert und in den Code Ihrer Funktion aufgenommen wurden. Wenn Sie Laufzeit-Hooks in einer separaten Datei oder einem Modul registrieren, müssen Sie sicherstellen, dass das Modul entweder direkt oder als Teil eines größeren Pakets in der Handler-Datei Ihrer Funktion importiert wird. Wenn die Datei oder das Modul nicht in den Funktionshandler importiert wird, ignoriert Lambda die Laufzeithaken.

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `beforeCheckpoint()`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit geladen werden und `afterRestore()`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

Schritt 1: Aktualisieren der Entwicklungskonfiguration

Fügen Sie die `org.crac`-Abhängigkeit zur Entwicklungskonfiguration hinzu. Im folgenden Beispiel wird Gradle verwendet. Beispiele für andere Entwicklungssysteme finden Sie in der Apache-Maven-Dokumentation.

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

Schritt 2: Aktualisieren des Lambda-Handlers

Der Lambda Event Function Handler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Weitere Informationen finden Sie unter Definieren Sie den Lambda-Ereignisfunktionshandler in Java.

Der folgende Beispielhandler zeigt, wie Code vor Checkpointing (`beforeCheckpoint()`) und nach der Wiederherstellung (`afterRestore()`) ausgeführt wird. Dieser Handler registriert auch die `Resource` im Laufzeit-verwalteten globalen `Context`.

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `beforeCheckpoint()`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit (JVM) geladen werden und `afterRestore()`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("After restore");
```

Der `Context` verwaltet nur ein `WeakReference` auf dem registrierten Objekt. Wenn für eine `Resource` eine Garbage Collection durchgeführt wird, werden Laufzeit-Hooks nicht ausgeführt. Ihr Code muss einen starken Verweis auf die `Resource` beibehalten, um sicherzustellen, dass der Laufzeit-Hook ausgeführt wird.

Hier sind zwei Beispiele für zu vermeidende Muster:

Example– Objekt ohne starke Referenz

```
Core.getGlobalContext().register( new MyResource() );
```

Example– Objekte anonymer Klassen

```
Core.getGlobalContext().register( new Resource() {

   @Override
   public void afterRestore(Context<? extends Resource> context) throws Exception {
    // ...
   }

   @Override
   public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
    // ...
   }

} );
```

Behalten Sie stattdessen eine starke Referenz bei. Im folgenden Beispiel wird die registrierte Ressource nicht von der Garbage Collection erfasst und Laufzeit-Hooks werden konsistent ausgeführt.

Example– Objekt mit starker Referenz

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Lambda SnapStart-Laufzeit-Hooks für Python

Sie können Laufzeit-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Funktion aus einem Snapshot fortsetzt. Python-Laufzeit-Hooks sind als Teil der Open-Source-Bibliothek Snapshot Restore für Python verfügbar, die in den von Python verwalteten Laufzeiten enthalten ist. Diese Bibliothek bietet zwei Dekoratoren, mit denen Sie Ihre Laufzeit-Hooks definieren können:

- `@register_before_snapshot`: Für Funktionen, die Sie ausführen möchten, bevor Lambda einen Snapshot erstellt.

- `@register_after_restore`: Für Funktionen, die Sie ausführen möchten, wenn Lambda eine Funktion aus einem Snapshot wieder aufnimmt.

Alternativ können Sie auch die folgenden Methoden verwenden, um Callables für Laufzeit-Hooks zu registrieren:

- `register_before_snapshot(func, *args, **kwargs)`
- `register_after_restore(func, *args, **kwargs)`

Registrierung und Ausführung von Laufzeit-Hooks

Die Reihenfolge, in der Lambda Ihre Laufzeit-Hooks ausführt, wird durch die Reihenfolge der Registrierung bestimmt:

- Vor dem Snapshot: Wird in umgekehrter Reihenfolge der Registrierung ausgeführt
- Nach dem Snapshot: Wird in Reihenfolge der Registrierung ausgeführt

Die Reihenfolge der Registrierung von Hooks zur Laufzeit hängt davon ab, wie Sie die Hooks definieren. Wenn Sie Dekoratoren (`@register_before_snapshot` und `@register_after_restore`) verwenden, folgt die Reihenfolge der Registrierung der Reihenfolge des Imports, der Definition oder der Ausführung in Ihrem Code. Wenn Sie mehr Kontrolle über die Registrierungsreihenfolge benötigen, verwenden Sie die `register_before_snapshot()`- und `register_after_restore()`-Methoden anstelle von Dekoratoren.

Vergewissern Sie sich, dass alle registrierten Hooks ordnungsgemäß importiert und in den Code Ihrer Funktion aufgenommen wurden. Wenn Sie Laufzeit-Hooks in einer separaten Datei oder einem Modul registrieren, müssen Sie sicherstellen, dass das Modul entweder direkt oder als Teil eines größeren Pakets in der Handler-Datei Ihrer Funktion importiert wird. Wenn die Datei oder das Modul nicht in den Funktionshandler importiert wird, ignoriert Lambda die Laufzeithaken.

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `@register_before_snapshot`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss

die Laufzeit geladen werden und `@register_after_restore`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

Beispiel

Der folgende Beispielhandler zeigt, wie Code vor Checkpointing (`@register_before_snapshot`) und nach der Wiederherstellung (`@register_after_restore`) ausgeführt wird.

```
from snapshot_restore_py import register_before_snapshot, register_after_restore

def lambda_handler(event, context):
    # Handler code

@register_before_snapshot
def before_checkpoint():
    # Logic to be executed before taking snapshots

@register_after_restore
def after_restore():
    # Logic to be executed after restore
```

Weitere Beispiele finden Sie unter Snapshot Restore für Python im AWS GitHub-Repository.

SnapStart Lambda-Runtime-Hooks für.NET

Sie können Runtime-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Event-Funktion aus einem Snapshot wieder aufnimmt. .NET-Runtime-Hooks sind als Teil des Amazon.Lambda.CorePakets verfügbar (Version 2.5.0 oder höher). Diese Bibliothek bietet zwei Methoden, mit denen Sie Ihre Laufzeit-Hooks definieren können:

- `RegisterBeforeSnapshot()`: Code, der vor der Snapshot-Erstellung ausgeführt werden soll
- `RegisterAfterSnapshot()`: Code, der nach der Wiederaufnahme einer Funktion aus einem Snapshot ausgeführt wird

> **Note**
>
> Wenn Sie das Lambda Annotations Framework für.NET verwenden, führen Sie ein Upgrade auf Amazon.Lambda.AnnotationsVersion 1.6.0 oder höher durch, um die Kompatibilität mit sicherzustellen. SnapStart

Registrierung und Ausführung von Laufzeit-Hooks

Registrieren Sie Ihre Hooks in Ihrem Initialisierungscode. Beachten Sie die folgenden Richtlinien, die auf dem Ausführungsmodell Ihrer Lambda-Funktion basieren:

- Für den Ansatz der ausführbaren Assemblierung müssen Sie Ihre Hooks registrieren, bevor Sie den Lambda-Bootstrap mit `RunAsync` starten.
- Für den Ansatz der Klassenbibliothek registrieren Sie Ihre Hooks im Konstruktor der Handler-Klasse.
- Registrieren Sie bei ASP.NET Core-Anwendungen Ihre Hooks, bevor Sie die Methode aufrufen. `WebApplications.Run`

Verwenden Sie die folgenden Methoden, um Runtime-Hooks für SnapStart in.NET zu registrieren:

```
Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
```

Wenn mehrere Hook-Typen registriert sind, wird die Reihenfolge, in der Lambda Ihre Laufzeit-Hooks ausführt, durch die Reihenfolge der Registrierung bestimmt:

- `RegisterBeforeSnapshot()`: Wird in umgekehrter Reihenfolge der Registrierung ausgeführt
- `RegisterAfterSnapshot()`: Wird in Reihenfolge der Registrierung ausgeführt

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `RegisterBeforeSnapshot()`-Laufzeit-Hooks werden auf das Zeitlimit des

Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit geladen werden und `RegisterAfterSnapshot()`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

Beispiel

Die folgende Beispielfunktion zeigt, wie Code vor Checkpointing (`RegisterBeforeSnapshot`) und nach der Wiederherstellung (`RegisterAfterRestore`) ausgeführt wird.

```
public class SampleClass
{
    public SampleClass()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
    }

    private ValueTask BeforeCheckpoint()
    {
        // Add logic to be executed before taking the snapshot
        return ValueTask.CompletedTask;
    }

    private ValueTask AfterCheckpoint()
    {
        // Add logic to be executed after restoring the snapshot
        return ValueTask.CompletedTask;
    }

    public APIGatewayProxyResponse FunctionHandler(APIGatewayProxyRequest request,
 ILambdaContext context)
    {
        // Add business logic

        return new APIGatewayProxyResponse
        {
            StatusCode = 200
        };
    }
}
```

Implementierung von SnapStart Hooks für Container-Images

-Übersicht

Bei Verwendung SnapStart mit einer Container-Image-Funktion muss die Runtime den Lebenszyklus der Funktion koordinieren und die Hooks vor dem Snapshot und nach der Wiederherstellung während der entsprechenden Lebenszyklusphasen aufrufen. Mit diesen Hooks können Sie benutzerdefinierte Logik (z. B. das Aktualisieren von Anmeldeinformationen oder das erneute Einsetzen von Zufallszahlengeneratoren) an bestimmten Punkten im Snapshot- und Wiederherstellungszyklus ausführen. Wenn Sie eine verwaltete Laufzeit verwenden, die unterstützt SnapStart wird, oder die entsprechenden Basis-Images für diese Laufzeiten (Java Version 11+, Python Version 3.12+ und .NET Version 8+), koordiniert Lambda den Lebenszyklus für Sie. Registrieren Sie Ihre Hooks über die unter beschriebene API. Implementieren Sie Code vor oder nach Snapshots der Lambda-Ereignisfunktion

Wenn Sie Ihre eigenen Basis-Container-Images, Runtime Interface Clients (RICs) oder die Basis-Images von Lambda für bereitgestellte.al2023 oder Ruby verwenden Node.js, folgen Sie zur Verwendung den Schritten auf dieser Seite. SnapStart

Voraussetzungen

Wenn Lambda eine Funktion aus einem Snapshot wiederherstellt, wird jeder während der Initialisierung definierte Status, wie Zufallszahlengeneratoren, eindeutige IDs und zwischengespeicherte Anmeldeinformationen, von allen Ausführungsumgebungen gemeinsam genutzt, die aus diesem Snapshot wiederhergestellt wurden. Überprüfen Sie vor der Verwendung SnapStart mit Ihrer Lambda-Ereignisfunktion, die auf Container-Images basiert, Umgang mit Einzigartigkeit mit Lambda SnapStart und stellen Sie sicher, dass die unter Verwenden von kryptografisch sicheren Pseudozufallszahlengeneratoren (CSPRNGs) beschriebenen Anforderungen erfüllt sind.

Nachdem Sie überprüft haben, ob die Anforderungen erfüllt sind, wählen Sie eine der folgenden beiden Optionen aus:

1. Option 1: Wenn Sie Hooks vor und nach der Wiederherstellung benötigen, um benutzerdefinierte Logik auszuführen, wenn der Snapshot Ihrer Funktion wieder aufgenommen wird, folgen Sie den Anweisungen im Abschnitt. Implementierung von Lebenszyklus-Hooks SnapStart
2. Option 2: Wenn Sie diese Hooks nicht benötigen, aktivieren Sie sie SnapStart für dieses Container-Image, indem Sie das folgende Label in Ihrem Dockerfile angeben:

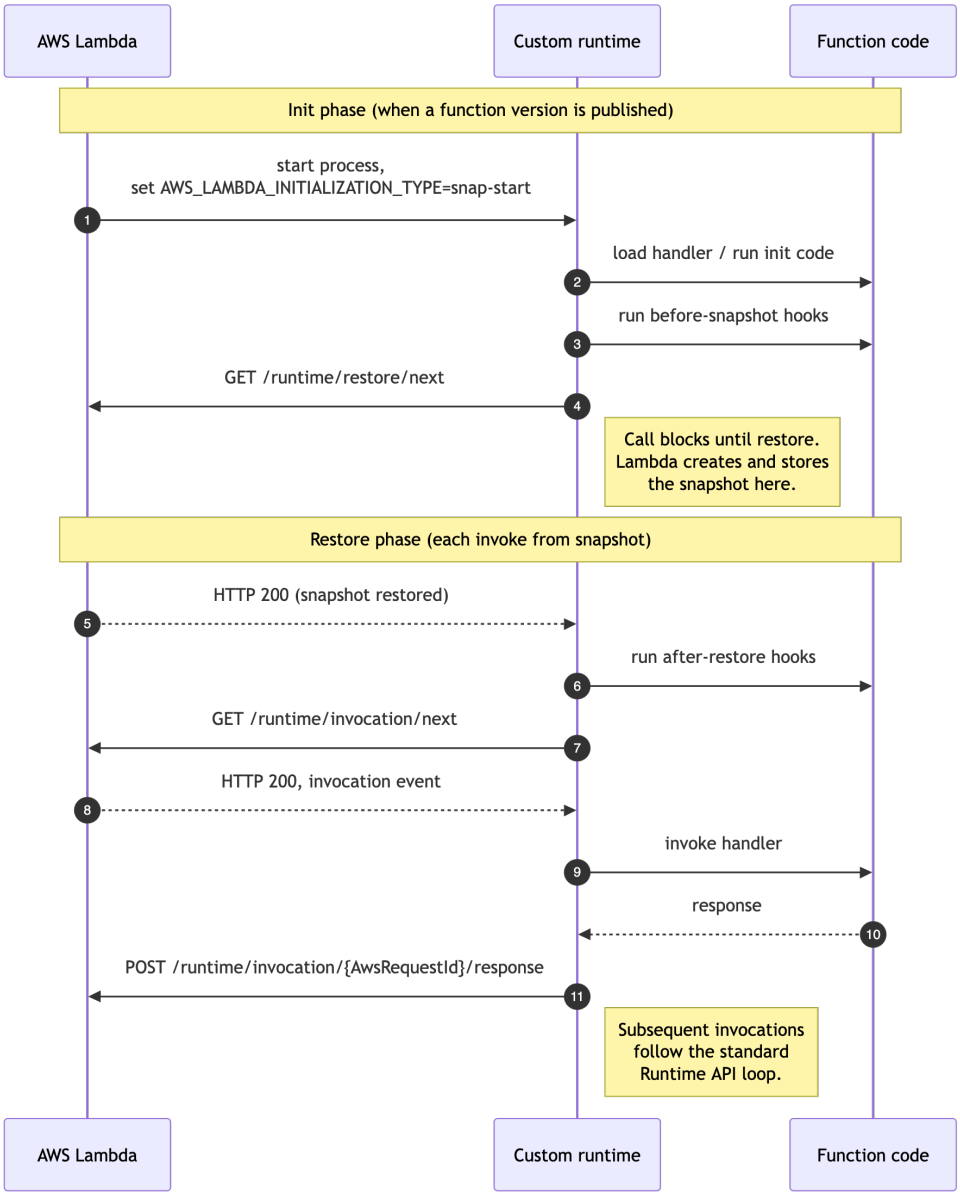
```
LABEL com.amazonaws.lambda.feature.snapstart="Allow"
```

Wenn Ihr Container-Image weder die `/restore/next` API implementiert noch das Label enthält, schlägt die Versionsveröffentlichung fehl.

> **Note**
>
> Diese Voraussetzungen sind nicht erforderlich, wenn Sie von Lambda verwaltete Basis-Images für Java (Version 11+), Python (Version 3.12+) und .NET (Version 8+) verwenden, da sie den Lebenszyklus bereits koordinieren und die SnapStart Eindeutigkeitsanforderungen bereitstellen.

Überblick über den Lebenszyklus

Das folgende Diagramm zeigt die Reihenfolge der Runtime-API-Aufrufe, die von einer SnapStart benutzerdefinierten Runtime erwartet werden. Alle Aufrufe sind Teil des Runtime-API-Vertrags, wohingegen die Aufrufe #3, #4, #5 und #6 spezifisch sind SnapStart.



Implementierung von Lebenszyklus-Hooks SnapStart

Gehen Sie wie folgt vor, um sie SnapStart mit Ihren Container-Image-Funktionen zu verwenden:

1. Führen Sie Before-Snapshot-Hooks aus und lösen Sie den Snapshot-Vorgang aus: Führen Sie als letzten Schritt des Initialisierungscodes Ihrer Funktion bei Bedarf die Before-Snapshot-Hooks aus und lösen Sie den Snapshot-Vorgang aus. Führen Sie diese Schritte nur aus, wenn die Option aktiviert SnapStart ist, indem Sie überprüfen, ob der Wert der Umgebungsvariablen auf gesetzt ist. `AWS_LAMBDA_INITIALIZATION_TYPE` `snap-start` Führen Sie Ihre registrierten Before-Snapshot-Hooks aus und rufen Sie sie dann auf, `GET /runtime/restore/next` um den Snapshot-Vorgang auszulösen. Wenn ein Before-Snapshot-Hook einen Fehler auslöst oder

zurückgibt, sendet die Runtime den Fehler an den Endpunkt. `/runtime/init/error` Sehen Sie sich das Pseudocode-Beispiel unten an:

```
# After all initialization code has finished:

   READ initialization_type FROM environment variable
 "AWS_LAMBDA_INITIALIZATION_TYPE"

   IF initialization_type IS "snap-start" THEN

       TRY
           EXECUTE registered before-snapshot hooks
       ON ERROR
           POST error to /runtime/init/error
               SET header  Lambda-Runtime-Function-Error-Type  TO
 <Category.Reason>
               SET body    TO  { errorMessage, errorType, stackTrace }
           EXIT process with non-zero code

       # Signal readiness for snapshot
       SEND GET request to /runtime/restore/next
       # The request blocks until Lambda restores the execution environment from
 the snapshot, then returns HTTP 200.

   END IF
```

> **Note**
>
> Die Hooks der Init-Phase und die Hooks vor dem Snapshot haben gemeinsam ein Timeout von. `max(function_timeout, 130 seconds)` Wenn dieses Limit überschritten wird, schlägt Lambda die PublishVersion Anfrage fehl. Außerdem ist `/runtime/restore/next` Call `/runtime/invocation/next` quasi ein blockierender Anruf. Es blockiert, bis Lambda die Ausführungsumgebung aus dem Snapshot wiederherstellt.

2. Führen Sie Hooks nach der Wiederherstellung aus und rufen Sie dann die Aufrufschleife auf. Wenn 200 `GET /runtime/restore/next` zurückgegeben wird, muss Ihre Runtime alle registrierten Hooks nach der Wiederherstellung ausführen, bevor sie zur Aufrufschleife übergeht. Wenn ein Hook nach der Wiederherstellung fehlschlägt, melden Sie den Fehler an. `/runtime/restore/error` Wenn die Hooks nach der Wiederherstellung abgeschlossen sind, rufen Sie

die Standard-Aufrufschleife auf, indem Sie aufrufen. `GET /runtime/invocation/next` Ab diesem Zeitpunkt ist das Verhalten identisch mit dem einer Funktion, die diese Funktion nicht verwendet. SnapStart Sehen Sie sich das Pseudocode-Beispiel unten an:

```
# After the snapshot has been restored
# (i.e., GET /runtime/restore/next has returned HTTP 200):

    TRY
        EXECUTE registered after-restore hooks
    ON ERROR
        POST error to /runtime/restore/error
            SET header  Lambda-Runtime-Function-Error-Type  TO  <Category.Reason>
            SET body    TO  { errorMessage, errorType, stackTrace }

# Proceed to the invoke loop
       GET /runtime/invocation/next
```

Fehlerbehandlung

Wenn ein Hook fehlschlägt, muss die Runtime den Fehler an den entsprechenden API-Endpunkt melden und den Prozess beenden. In der folgenden Tabelle ist das Verhalten für jede Phase zusammengefasst:

Phase	Fehler beim API-Endpunkt	Was passiert bei einem Fehler
Init//vor dem Snapshot	`POST /runtime/init/error`	Lambda schlägt die PublishVersion Anfrage fehl. Beenden Sie den Prozess.
After-restore	`POST /runtime/restore/error`	Lambda schlägt beim In-Flight-Aufruf fehl und zerstört die Ausführungsumgebung. Beenden Sie den Prozess.

Legen Sie für beide API-Endpunkte den `Lambda-Runtime-Function-Error-Type` Header auf einen Wert im Format fest `<Category.Reason>` (z. B. `Runtime.BeforeSnapshotError` oder`Runtime.AfterRestoreError`). Fügen Sie einen Fehlertext mit `errorMessageerrorType`, und ein optionales `stackTrace` ein.

Die vollständige API-Endpunktspezifikation und die Antwortcodes finden Sie unter Initialisierungsfehler und Wiederherstellungsfehler (gilt nur für SnapStart) in der Runtime-API-Referenz.

Überwachung für Lambda SnapStart

Sie können Ihre SnapStart Lambda-Funktionen mit Amazon CloudWatch, AWS X-Ray, und dem Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API überwachen.

> **Note**
>
> Die `AWS_LAMBDA_LOG_STREAM_NAME` Umgebungsvariablen `AWS_LAMBDA_LOG_GROUP_NAME` und sind in SnapStart Lambda-Funktionen nicht verfügbar.

Verständnis des Protokollierungs- und Abrechnungsverhaltens mit SnapStart

Es gibt einige Unterschiede beim CloudWatch Logstream-Format für SnapStart Funktionen:

- Initialisierungsprotokolle – Wenn eine neue Ausführungsumgebung erstellt wird, enthält `REPORT` das Feld `Init Duration` nicht. Das liegt daran, dass Lambda SnapStart Funktionen initialisiert, wenn Sie eine Version erstellen, und nicht beim Funktionsaufruf. Bei SnapStart Funktionen befindet sich das `Init Duration` Feld im Datensatz. `INIT_REPORT` Dieser Datensatz zeigt die Dauer von Init-Phase, einschließlich der Dauer aller `beforeCheckpoint`-Laufzeit-Hooks an.
- Aufrufprotokolle – Wenn eine neue Ausführungsumgebung erstellt wird, enthält `REPORT` die Felder `Restore Duration` und `Billed Restore Duration`:
  - `Restore Duration`: Die Zeit, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und ggf. Laufzeit-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird in `Restore Duration` erfasst.
  - `Billed Restore Duration`: Die Zeit, die Lambda benötigt, um die Laufzeit zu laden und ggf. Laufzeit-Hooks nach der Wiederherstellung auszuführen.

> **Note**
>
> Wie bei allen Lambda-Ereignisfunktionen fallen Gebühren für die Dauer von Code an, der im Funktionshandler ausgeführt wird. Bei SnapStart Funktionen fallen Dauergebühren auch für Initialisierungscode an, der außerhalb des Handlers deklariert wurde, für die Zeit, die zum Laden der Laufzeit benötigt wird, und für jeden Code, der in einem Runtime-Hook ausgeführt wird.

Die Kaltstartdauer ist die Summe von `Restore Duration` + `Duration`.

Das folgende Beispiel ist eine Lambda Insights-Abfrage, die die Latenzperzentile für Funktionen zurückgibt. SnapStart Weitere Informationen zu Lambda-Insights-Abfragen finden Sie unter Beispiel für einen Workflow, der Abfragen zur Fehlerbehebung bei einer Ereignisfunktion verwendet.

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

X-Ray aktive Ablaufverfolgung für SnapStart

Sie können es verwenden X-Ray, um Anfragen an SnapStart Lambda-Funktionen zu verfolgen. Bei den X-Ray Untersegmenten für SnapStart Funktionen gibt es einige Unterschiede:

- Es gibt kein `Initialization` Untersegment für SnapStart Funktionen.
- Das Teilsegment `Restore` zeigt die Zeit an, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und vorhandene Laufzeit-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird im `Restore`-Untersegment erfasst. Die Zeit, die Sie außerhalb der microVM für die Wiederherstellung eines Snapshots aufwenden, wird Ihnen nicht in Rechnung gestellt.

Telemetrie-API-Ereignisse für SnapStart

Lambda sendet die folgenden SnapStart Ereignisse anTelemetrie-API:

- `Plattform. Neustart wiederherstellen` – Zeigt die Zeit an, zu der die `Restore`-Phase gestartet wurde.

- `plattform.wiederherstellen RuntimeDone` – Zeigt an, ob die `Restore`-Phase erfolgreich war. Lambda sendet diese Nachricht, wenn die Laufzeit eine `restore/next`-Laufzeit-API-Anfrage sendet. Es gibt drei mögliche Status: erfolgreich, fehlgeschlagen und Timeout.
- `Platform.RestoreReport` – Zeigt an, wie lange die `Restore`-Phase gedauert hat und wie viele Millisekunden Ihnen während dieser Phase in Rechnung gestellt wurden.

Amazon-API-Gateway- und Funktions-URL-Metriken

Wenn Sie mit API Gateway eine Web-API erstellen, können Sie die IntegrationLatencyMetrik verwenden, um die End-to-End-Latenz zu messen (die Zeit zwischen der Weiterleitung einer Anfrage an das Backend durch API Gateway und dem Empfang einer Antwort vom Backend).

Wenn Sie eine Lambda-Funktions-URL verwenden, können Sie die UrlRequestLatencyMetrik verwenden, um die End-to-End-Latenz zu messen (die Zeit zwischen dem Empfang einer Anfrage durch die Funktions-URL und dem Zeitpunkt, an dem die Funktions-URL eine Antwort zurückgibt).

Sicherheitsmodell für Lambda SnapStart

AWS Lambda SnapStart unterstützt Verschlüsselung im Ruhezustand. Lambda verschlüsselt Snapshots mit einem. AWS KMS key Standardmäßig verwendet Lambda eine Von AWS verwalteter Schlüssel. Wenn dieses Standardverhalten für Ihren Workflow geeignet ist, müssen Sie nichts weiter einrichten.

Andernfalls können Sie die `--kms-key-arn` Option im Befehl create-function oder update-function-configuration verwenden, um einen vom Kunden verwalteten Schlüssel bereitzustellen. AWS KMS Sie können dies tun, um die Rotation des KMS-Schlüssels zu steuern oder um die Anforderungen Ihrer Organisation für die Verwaltung von KMS-Schlüsseln zu erfüllen. Für vom Kunden verwaltete Schlüssel fallen Standard- AWS KMS -Gebühren an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Wenn es sich bei Ihrer Event-Funktion auch um eine dauerhafte Funktion handelt, ist der vom Kunden verwaltete Schlüssel, der SnapStart Snapshots verschlüsselt, unabhängig vom vom Kunden verwalteten Schlüssel, der dauerhafte Ausführungsdaten verschlüsselt. Weitere Informationen finden Sie unter Verschlüsseln AWS Lambda dauerhafte Ausführungsdaten.

Wenn Sie eine SnapStart Funktion oder Funktionsversion löschen, schlagen alle `Invoke` Anfragen an diese Funktion oder Funktionsversion fehl. Lambda entfernt alle Ressourcen, die mit gelöschten Snapshots verknüpft sind, gemäß der Allgemeinen Datenschutzverordnung (DSGVO).

Maximieren Sie die Lambda-Leistung SnapStart

Themen

- Leistungsoptimierung
- Bewährte Praktiken für das Netzwerk

Leistungsoptimierung

Beachten Sie die folgenden Empfehlungen zur Codeoptimierung für Ihre Runtime SnapStart, um die Vorteile von zu maximieren.

> **Note**
>
> SnapStart funktioniert am besten, wenn es mit Funktionsaufrufen in großem Maßstab verwendet wird. Funktionen, die selten aufgerufen werden, weisen möglicherweise nicht dieselben Leistungsverbesserungen auf.

Java

Um die Vorteile von zu maximieren SnapStart, empfehlen wir, Abhängigkeiten vorab zu laden und Ressourcen, die zur Startlatenz beitragen, in Ihrem Initialisierungscode statt im Eventfunktionshandler zu initialisieren. Dadurch wird die Latenz, die mit dem Laden schwerer Klassen verbunden ist, aus dem Aufrufpfad entfernt, wodurch die Startleistung mit optimiert wird. SnapStart

Wenn Sie Abhängigkeiten oder Ressourcen nicht während der Initialisierung vorladen können, empfehlen wir Ihnen, sie mit Dummy-Aufrufen vorzuladen. Aktualisieren Sie dazu den Code des Funktionshandlers, wie im folgenden Beispiel aus der Funktion pet store im AWS GitHub Labs-Repository gezeigt.

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
          handler.onStartup(servletContext -> {
```

```
            FilterRegistration.Dynamic registration =
servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
            registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
false, "/*");
        });

        // Send a fake Amazon API Gateway request to the handler to load classes
ahead of time
        ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
        identity.setApiKey("foo");
        identity.setAccountId("foo");
        identity.setAccessKey("foo");

        AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
        reqCtx.setPath("/pets");
        reqCtx.setStage("default");
        reqCtx.setAuthorizer(null);
        reqCtx.setIdentity(identity);

        AwsProxyRequest req = new AwsProxyRequest();
        req.setHttpMethod("GET");
        req.setPath("/pets");
        req.setBody("");
        req.setRequestContext(reqCtx);

        Context ctx = new TestContext();
        handler.proxy(req, ctx);


    } catch (ContainerInitializationException e) {
        // if we fail here. We re-throw the exception to force another cold start
        e.printStackTrace();
        throw new RuntimeException("Could not initialize Spring framework", e);
    }
 }
```

Python

Um die Vorteile von zu maximieren SnapStart, konzentrieren Sie sich auf eine effiziente Codeorganisation und Ressourcenverwaltung innerhalb Ihrer Python-Funktionen. Als allgemeine Richtlinie gilt, dass Sie während der Initialisierungsphase umfangreiche Rechenaufgaben ausführen sollten. Mit diesem Ansatz werden zeitaufwändige Operationen aus dem Aufrufpfad

herausgenommen, was die Gesamtleistung der Funktion verbessert. Um diese Strategie wirksam umzusetzen, empfehlen wir die folgenden bewährten Verfahren:

- Import von Abhängigkeiten außerhalb des Funktionshandlers.
- Erstellen Sie `boto3`-Instances außerhalb des Handlers.
- Initialisieren Sie statische Ressourcen oder Konfigurationen, bevor der Handler aufgerufen wird.
- Erwägen Sie die Verwendung eines Laufzeit-Hooks vor dem Snapshot für ressourcenintensive Aufgaben wie das Herunterladen externer Dateien, das Vorladen von Frameworks wie Django oder das Laden von Machine-Learning-Modellen.

Example— Optimieren Sie die Python-Event-Funktion für SnapStart

```
# Import all dependencies outside of Lambda handler
from snapshot_restore_py import register_before_snapshot
import boto3
import pandas
import pydantic

# Create S3 and SSM clients outside of Lambda handler
s3_client = boto3.client("s3")

# Register the function to be called before snapshot
@register_before_snapshot
def download_llm_models():
    # Download an object from S3 and save to tmp
    # This files will persist in this snapshot
    with open('/tmp/FILE_NAME', 'wb') as f:
        s3_client.download_fileobj('amzn-s3-demo-bucket', 'OBJECT_NAME', f)
    ...

def lambda_handler(event, context):
    ...
```

.NET

Um die Just-in-Time-Kompilierung (JIT) und das Laden von Assemblys zu reduzieren, sollten Sie erwägen, Ihren Event-Funktionshandler von einem Runtime-Hook aus aufzurufen. `RegisterBeforeCheckpoint` Aufgrund der Funktionsweise der mehrstufigen Kompilierung von.NET erzielen Sie optimale Ergebnisse, wenn Sie den Handler mehrmals aufrufen, wie im folgenden Beispiel gezeigt.

> **Important**
>
> Vergewissern Sie sich, dass Ihr Dummy-Funktionsaufruf keine unbeabsichtigten Nebeneffekte hervorruft, wie z. B. das Auslösen von Geschäftstransaktionen.

Example

```
public class Function
{
    public Function()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(FunctionWarmup);
    }

    // Warmup method that calls the function handler before snapshot to warm up
 the .NET code and runtime.
    // This speeds up future cold starts after restoring from a snapshot.

    private async ValueTask FunctionWarmup()
    {
        var request = new APIGatewayProxyRequest
        {
            Path = "/heathcheck",
            HttpMethod = "GET"
        };

        for (var i = 0; i < 10; i++)
        {
            await FunctionHandler(request, null);
        }
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        //
        // Process HTTP request
        //

        var response = new APIGatewayProxyResponse
        {
            StatusCode = 200
```

```
        };

        return await Task.FromResult(response);
    }
}
```

Bewährte Praktiken für das Netzwerk

Der Status der Verbindungen, die Ihre Event-Funktion während der Initialisierungsphase herstellt, ist nicht garantiert, wenn Lambda Ihre Funktion von einem Snapshot aus wieder aufnimmt. In den meisten Fällen werden Netzwerkverbindungen, die ein AWS SDK herstellt, automatisch wieder aufgenommen. Für andere Verbindungen empfehlen wir die folgenden bewährten Methoden.

Re-establish Netzwerkverbindungen

Stellen Sie Ihre Netzwerkverbindungen immer wieder her, wenn Ihre Funktion von einem Snapshot fortgesetzt wird. Wir empfehlen, dass Sie die Netzwerkverbindungen im Funktionshandler wiederherstellen. Alternativ können Sie ein After-Restore-Laufzeit-Hook verwenden.

Verwenden Sie den Hostnamen nicht als eindeutige ID der Ausführungsumgebung

Wir raten davon ab, `hostname` zu verwenden, um Ihre Ausführungsumgebung als eindeutigen Knoten oder Container in Ihren Anwendungen zu identifizieren. Bei SnapStart wird ein einzelner Snapshot als Ausgangszustand für mehrere Ausführungsumgebungen verwendet. Alle Ausführungsumgebungen geben denselben `hostname`-Wert für `InetAddress.getLocalHost()` (Java), `socket.gethostname()` (Python) und `Dns.GetHostName()` (.NET) zurück. Für Anwendungen, die eine eindeutige Identität oder einen eindeutigen `hostname`-Wert der Ausführungsumgebung erfordern, empfehlen wir, im Funktionshandler eine eindeutige ID zu generieren. Oder verwenden Sie einen After-Restore-Laufzeit-Hook, um eine eindeutige ID zu generieren und verwenden Sie dann die eindeutige ID für die Ausführungsumgebung.

Vermeiden der Bindung von Verbindungen an feste Quellports

Wir empfehlen, Netzwerkverbindungen nicht an feste Quellports zu binden. Wenn eine Funktion nach einem Snapshot wieder aufgenommen wird, werden die Verbindungen neu aufgebaut, und Netzwerkverbindungen, die an einen festen Quellport gebunden sind, können fehlschlagen.

Vermeiden der Verwendung von Java-DNS-Cache

Lambda-Ereignisfunktionen speichern bereits DNS-Antworten im Cache. Wenn Sie einen anderen DNS-Cache mit verwenden SnapStart, kann es zu Verbindungstimeouts kommen, wenn die Funktion von einem Snapshot aus wieder aufgenommen wird.

Die `java.util.logging.Logger`-Klasse kann indirekt den JVM-DNS-Cache aktivieren. Um die Standardeinstellungen zu überschreiben, setzen Sie networkaddress.cache.ttl auf 0, bevor Sie `logger` initialisieren. Beispiel:

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

Um `UnknownHostException`-Ausfälle in der Java 11-Laufzeit zu vermeiden, empfehlen wir, `networkaddress.cache.negative.ttl` auf 0 zu setzen. In Java 17 und späteren Laufzeiten ist dieser Schritt nicht erforderlich. Sie können diese Eigenschaft für eine Lambda-Ereignisfunktion mit der `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` Umgebungsvariablen festlegen.

Das Deaktivieren des JVM-DNS-Cache deaktiviert nicht das verwaltete DNS-Caching von Lambda.

Behebung SnapStart von Fehlern für Lambda-Ereignisfunktionen

Auf dieser Seite werden häufig auftretende Probleme behandelt, die bei der Verwendung von Lambda auftreten SnapStart, darunter Fehler bei der Snapshot-Erstellung, Timeoutfehler und interne Dienstfehler.

SnapStartNotReadyException

Fehler: Beim Aufrufen der Operation Invoke20150331 ist ein Fehler aufgetreten (SnapStartNotReadyException): Lambda initialisiert Ihre Event-Funktion. Sie ist aufrufbereit, sobald der Status Ihrer Funktion auf AKTIV gesetzt wird.

Häufige Ursachen

Dieser Fehler tritt auf, wenn Sie versuchen, eine Funktionsversion aufzurufen, die sich im `Inactive` Status befindet. Die Version Ihrer Funktion wird zu `Inactive`, wenn sie 14 Tage lang nicht mehr aufgerufen wurde oder wenn Lambda die Ausführungsumgebung periodisch recycelt

Auflösung

Warten Sie, bis die Funktionsversion den `Active`-Status erreicht hat, und rufen Sie es dann erneut auf.

SnapStartTimeoutException

Problem: Sie erhalten eine, `SnapStartTimeoutException` wenn Sie versuchen, eine Funktionsversion aufzurufen. SnapStart

Häufige Ursache

In der Wiederherstellungsphase stellt Lambda die Java-Laufzeit wieder her und führt alle Laufzeit-Hooks nach der Wiederherstellung aus. Wenn ein Laufzeit-Hook nach der Wiederherstellung länger als 10 Sekunden läuft, wird die `Restore`-Phase unterbrochen und Sie erhalten einen Fehler, wenn Sie versuchen, die Funktion aufzurufen. Probleme mit der Netzwerkverbindung und den Anmeldeinformationen können auch zu `Restore`-Phase-Timeouts führen.

Auflösung

Suchen Sie in den CloudWatch Protokollen der Funktion nach Timeoutfehlern, die während der Wiederherstellungsphase aufgetreten sind. Stellen Sie sicher, dass alle Hooks nach der Wiederherstellung in weniger als 10 Sekunden abgeschlossen sind.

Example CloudWatch protokollieren

```
{ "cause": "Lambda couldn't restore the snapshot within the timeout limit. (Service:
 Lambda, Status Code: 408, Request ID: 11a222c3-410f-427c-ab22-931d6bcbf4f2)", "error":
 "Lambda.SnapStartTimeoutException"}
```

500 Interner Dienstfehler

Fehler: Lambda konnte keinen neuen Snapshot erstellen, weil Sie das Limit für die gleichzeitige Erstellung von Snapshots erreicht haben.

Häufige Ursache

Ein 500-Fehler ist ein interner Fehler innerhalb des Lambda-Service selbst und kein Problem mit Ihrer Funktion oder Ihrem Code. Diese Fehler treten häufig sporadisch auf.

Auflösung

Versuchen Sie erneut, die Funktionsversion zu veröffentlichen.

401 Nicht autorisiert

Fehler: Falsches Sitzungstoken oder Header-Schlüssel

Häufige Ursache

Dieser Fehler tritt auf, wenn Sie den AWS Systems Manager Parameter Store und die AWS Secrets Manager Erweiterung mit Lambda SnapStart verwenden. Wenn Ihre Funktion verwendet SnapStart, ruft sie Anmeldeinformationen von einem Container-Endpunkt ab, und Lambda legt die `AWS_SESSION_TOKEN` Umgebungsvariable nicht fest. Wenn Ihre Funktion den `X-Aws-Parameters-Secrets-Token` Header aus dieser Umgebungsvariablen festlegt, ist der Header leer und die Erweiterung lehnt die Anfrage ab.

Auflösung

Um diesen Fehler zu beheben, verwenden Sie die AWS SDK-Anmeldeinformationsanbieterkette, um das Sitzungstoken abzurufen, anstatt die `AWS_SESSION_TOKEN` Umgebungsvariable direkt zu lesen. Setzen Sie den `X-Aws-Parameters-Secrets-Token` Header auf das von der Anbieterkette zurückgegebene Token. Die Anbieterkette für Anmeldeinformationen gibt in allen Initialisierungsmodi das richtige Sitzungstoken zurück, einschließlich. SnapStart Einen Beispielcode finden Sie unter Verwenden Sie Secrets Manager Manager-Geheimnisse in Lambda-Ereignisfunktionen.

UnknownHostException (Java)

Fehler: HTTP-Anfrage kann nicht ausgeführt werden: Das Zertifikat für `abc.us-east-1.amazonaws.com` stimmt mit keinem der alternativen Namen des Betreffs überein.

Häufige Ursache

Lambda-Ereignisfunktionen speichern bereits DNS-Antworten im Cache. Wenn Sie einen anderen DNS-Cache mit verwenden SnapStart, kann es zu Verbindungstimeouts kommen, wenn die Funktion von einem Snapshot aus wieder aufgenommen wird.

Auflösung

Um `UnknownHostException`-Ausfälle in der Java 11-Laufzeit zu vermeiden, empfehlen wir, `networkaddress.cache.negative.ttl` auf 0 zu setzen. In Java 17 und späteren Laufzeiten ist dieser Schritt nicht erforderlich. Sie können diese Eigenschaft für eine Lambda-Ereignisfunktion mit der `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` Umgebungsvariablen festlegen.

Fehler bei der Erstellung von Snapshots

Fehler: Ihre AWS Lambda SnapStart Funktion konnte nicht aufgerufen werden. Wenn dieser Fehler weiterhin besteht, überprüfen Sie die CloudWatch Protokolle Ihrer Funktion auf Initialisierungsfehler.

Auflösung

Überprüfen Sie die CloudWatch Amazon-Logs Ihrer Funktion auf Timeouts vor dem Checkpoint-Runtime-Hook. Sie können auch versuchen, eine neue Funktionsversion zu veröffentlichen, wodurch das Problem manchmal behoben werden kann.

Latenzzeit bei der Snapshot-Erstellung

Problem: Wenn Sie eine neue Funktionsversion veröffentlichen, bleibt die Funktion lange im `Pending` Status.

Häufige Ursache

Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist.

Wenn Ihre Funktion mit einer VPC verbunden ist, muss Lambda möglicherweise auch Netzwerkschnittstellen erstellen, bevor die Funktion `Active` wird. Wenn Sie versuchen, die Funktionsversion aufzurufen, während die Funktion `Pending` ist, erhalten Sie möglicherweise eine 409 `ResourceConflictException`. Wenn die Funktion über einen Amazon-API-Gateway-Endpunkt aufgerufen wird, erhalten Sie möglicherweise einen 500-Fehler im API Gateway.

Auflösung

Warten Sie mindestens 15 Minuten, bis die Funktionsversion initialisiert ist, bevor Sie sie aufrufen.

Isolierung von Mietern

Verwenden Sie den Mandantenisolationsmodus, wenn Sie eine isolierte Anforderungsverarbeitung für einzelne Endbenutzer oder Mandanten benötigen, die eine Lambda-Ereignisfunktion aufrufen, die früher als Lambda-Funktion bezeichnet wurde. Diese Funktion vereinfacht die Erstellung von Mehrmandantenanwendungen, die mandantenspezifischen Code oder Daten verarbeiten, wie z. B. SaaS-Plattformen für die Workflow-Automatisierung oder Codeausführung, da die Verwaltung mandantenspezifischer Funktionsressourcen und die Anforderungsrouting-Logik entfällt.

Multi-tenant Für Anwendungen gelten strenge Isolationsanforderungen, wenn Code ausgeführt oder Daten für einzelne Mandanten oder Endbenutzer verarbeitet werden. Im Tenant-Isolationsmodus verwendet Lambda eine vom Kunden angegebene Mandanten-ID, um Anfragen an die zugrunde liegenden Funktionsausführungsumgebungen weiterzuleiten. Dadurch wird sichergestellt, dass die Ausführungsumgebungen einer Funktion nur für Aufrufe vom angegebenen Endbenutzer oder Mandanten verwendet werden. Die Funktionsausführungsumgebungen von Lambda nutzen die Firecracker-Virtualisierung, um Arbeitslasten zu isolieren.

Wenn eine Ereignisfunktion, die den Tenant-Isolationsmodus verwendet, einen Aufruf mit einer Mandanten-ID empfängt, versucht Lambda zunächst, eine verfügbare Ausführungsumgebung zu finden, die dieser Mandanten-ID zugeordnet ist. Wenn keine Ausführungsumgebungen vorhanden sind, erstellt Lambda eine neue Ausführungsumgebung und weist diesem Mandanten eine neue Ausführungsumgebung zu. Wenn Funktionsaufrufe mit der angegebenen Mandanten-ID immer größer werden, sucht Lambda nach Bedarf nach neuen Ausführungsumgebungen oder erstellt sie.

Informationen zur Verwendung der Mandantenisolierung finden Sie unter. the section called “Aktivierung der Mandantenisolierung” Informationen zum Zugriff auf den Mandantenkontext in Ihrem Code finden Sie unterthe section called “Zugriff im Code”. Informationen zur Überwachung von Funktionen, die vom Mandanten isoliert sind, finden Sie unter. the section called “Überwachen” Hilfe zur Lösung von Problemen finden Sie unter. the section called “Fehlerbehebung”

Themen

- Wann sollte der Mandantenisolationsmodus verwendet werden
- Unterstützte Funktionen und Einschränkungen
- Unterstützt AWS-Regionen
- Überlegungen
- Preisgestaltung
- Isolationsmodus
- Aktivierung der Mandantenisolierung für Lambda-Ereignisfunktionen
- Aufrufen von Lambda-Ereignisfunktionen mit Mandantenisolierung
- Zugreifen auf die Mandanten-ID im Lambda-Ereignisfunktionscode
- Überwachung von Lambda-Ereignisfunktionen mit Mandantenisolierung
- Problembehandlung bei der Mandantenisolierung für Lambda Event Functions

Wann sollte der Mandantenisolationsmodus verwendet werden

Verwenden Sie den Mandantenisolationsmodus, wenn Sie mehrere Endbenutzer oder Mandanten mit einer einzigen Lambda-Ereignisfunktion bedienen müssen, und stellen Sie gleichzeitig sicher, dass die Ausführungsumgebungen, die zur Verarbeitung von Aufrufen für einzelne Mandanten verwendet werden, voneinander isoliert bleiben. Diese strikte Isolierung der Ausführungsumgebungen ist für Mehrmandantenanwendungen erforderlich, die:

- Vom Endbenutzer bereitgestellten Code ausführen: Durch die Beibehaltung separater Ausführungsumgebungen für einzelne Mandanten können die Auswirkungen der Ausführung von vom Benutzer bereitgestelltem Code, der falsch oder bösartig sein könnte, begrenzt werden.
- Verarbeitung mandantenspezifischer Daten: Durch die Beibehaltung separater Ausführungsumgebungen für einzelne Mandanten kann verhindert werden, dass vertrauliche mandantenspezifische Daten an andere Mandanten weitergegeben werden.

Durch mehrere Aufrufanfragen desselben Mandanten kann dieselbe Funktionsausführungsumgebung wiederverwendet werden, wodurch Kaltstarts reduziert und die Reaktionszeiten für latenzempfindliche Anwendungen verbessert werden.

Unterstützte Funktionen und Einschränkungen

Der Mandantenisolationsmodus wird bei Ereignisfunktionen, die Funktions-URLs, bereitgestellte Parallelität verwenden, oder SnapStartbei Ereignisfunktionen, die auf verwalteten Lambda-Instanzen ausgeführt werden, nicht unterstützt. Sie können Anfragen an eine vom Mandanten isolierte Funktion senden, indem Sie synchrone Aufrufe, asynchrone Aufrufeoder Amazon API Gateway als Event-Trigger verwenden.

Unterstützt AWS-Regionen

Der Mandantenisolationsmodus wird in allen kommerziellen Regionen außer Asien-Pazifik (Neuseeland) unterstützt.

Überlegungen

Beachten Sie bei der Verwendung der Mandantenisolierung mit Ihren Lambda-Ereignisfunktionen Folgendes:

- Unveränderliche Konfiguration: Die Mandantenisolierung ist eine unveränderliche Funktionseigenschaft. Sie kann nur aktiviert werden, wenn eine Funktion erstellt wird.
- Erforderlicher Tenant-ID-Parameter: Ereignisfunktionen, die den Tenant-Isolationsmodus verwenden, müssen mit einem Parameter aufgerufen werden. `tenant-id` Wenn Sie diesen Parameter weglassen, schlagen Funktionsaufrufe fehl.
- Die Ausführungsrolle gilt für alle Mandanten: Aufrufe von allen Mandanten verwenden die in der Ausführungsrolle Ihrer Lambda-Ereignisfunktion definierten Berechtigungen.
- Parallelität: Es gibt keine Änderungen an der Parallelität oder dem Skalierungsverhalten Ihrer Event-Funktion, wenn Sie die Mandantenisolierung verwenden. Lambda legt eine Obergrenze von 2.500 mandantenisolierten Ausführungsumgebungen (aktiv oder inaktiv) für jeweils 1.000 gleichzeitige Ausführungen fest, die für Ihre Lambda Event Function konfiguriert sind.

Preisgestaltung

Ihnen wird eine Gebühr berechnet, wenn Lambda eine neue, vom Mandanten isolierte Ausführungsumgebung erstellt. Der Preis hängt von der Speichermenge ab, die Sie Ihrer Event Function zuweisen, und von der CPU-Architektur, die Sie verwenden. Weitere Informationen finden Sie unter AWS Lambda Preise.

Isolationsmodus

In der folgenden Tabelle werden die Unterschiede zwischen Lambda-Ereignisfunktionen mit und ohne Mandantenisolierung beschrieben.

Feature	Mit Mandantenisolierung	Ohne Isolierung der Mieter
Art der Isolierung	Tenant-level Isolierung	Function-level Isolierung
Wiederverwendung der Umwelt	Ausführungsumgebungen werden niemals mandantenübergreifend wiederverwendet	Ausführungsumgebungen können bei Aufrufen derselben Funktion wiederverwendet werden
Datenisolierung	Auf Daten von anderen Mandanten kann nicht zugegriffen werden	Auf Daten aus früheren Aufrufen derselben Funktionsversion kann möglicherweise zugegriffen werden
Kalte Starts	Mehr Kaltstarts aufgrund von mieterspezifischen Umgebungen	Weniger Kaltstarts aufgrund der Wiederverwendung der Umgebung
Preisgestaltung	Zusätzliche Gebühr neben den Standard-Lambda-Preisen	Lambda-Standardpreise

Aktivierung der Mandantenisolierung für Lambda-Ereignisfunktionen

Um den Tenant-Isolationsmodus zu aktivieren, erstellen Sie eine neue Lambda-Ereignisfunktion. Sie können die Mandantenisolierung für bestehende Eventfunktionen nicht aktivieren.

Themen

- Aktivierung der Mandantenisolierung (Konsole)
- Aktivierung der Mandantenisolierung (AWS CLI)
- Aktivierung der Mandantenisolierung (API)
- Aktivierung der Mandantenisolierung (CloudFormation)

Aktivierung der Mandantenisolierung (Konsole)

So erstellen Sie eine Lambda-Ereignisfunktion mit der Konsole

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie im Bereich Grundlegende Informationen als Funktionsname **`image-analysis`** ein.
5. Wählen Sie für Runtime eine der unterstützten Lambda-Laufzeiten aus.
6. Wählen Sie unter Zusätzliche Konfigurationen, Tenant-Isolationsmodus, die Option Aktivieren aus.
7. Überprüfen Sie Ihre Einstellungen und wählen Sie Create Event Function aus.

Aktivierung der Mandantenisolierung (AWS CLI)

Ereignisfunktion mit Mandantenisolierung erstellen

Wenn Sie eine neue Event-Funktion mit der CLI erstellen, fügen Sie die `--tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'` Option zu Ihrer Create-Function-Anfrage hinzu. Beispiel:

```
aws lambda create-function \
    --function-name image-analysis \
    --runtime nodejs24.x \
    --zip-file fileb://image-analysis-function.zip \
```

```
    --handler image-analysis-function.handler \
    --role arn:aws:iam:123456789012:role/execution-role \
    --tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'
```

Aktivierung der Mandantenisolierung (API)

So aktivieren Sie die Mandantenisolierung mithilfe der Lambda-API

1. Erstellen Sie eine neue Ereignisfunktion mit aktivierter Mandantenisolierung, indem Sie die CreateFunctionAPI-Aktion mit dem `TenancyConfig` Parameter verwenden.
2. Vergewissern Sie sich mithilfe der GetFunctionConfigurationAktion, dass die Mandantenisolierung für die Ereignisfunktion aktiviert ist. Wenn die Antwort darauf hinweistPER_TENANT, dass dies der Fall `TenantIsolationMode` ist, ist die Mandantenisolierung für die Ereignisfunktion aktiviert:

   ```
   "TenancyConfig": {
           "TenantIsolationMode": "PER_TENANT"
       }
   ```

Rufen Sie die Funktionsversion mit der Aufruf-Aktion auf. Weitere Informationen finden Sie unter Aufrufen von Lambda-Ereignisfunktionen mit Mandantenisolierung.

Aktivierung der Mandantenisolierung (CloudFormation)

Die folgende CloudFormation Vorlage erstellt eine neue Lambda-Ereignisfunktion mit aktivierter Mandantenisolierung:

```
MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-sample-python-lambda
      Runtime: python3.14
      Role: !GetAtt LambdaExecutionRole.Arn
      Handler: index.lambda_handler
      TenancyConfig:
        TenantIsolationMode: PER_TENANT
      Code:
        ZipFile: |
          import json
```

```
        def lambda_handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps(f'Hello from Lambda! Tenant-ID:
{context.tenant_id}')
            }
    Timeout: 10
    MemorySize: 128
```

Aufrufen von Lambda-Ereignisfunktionen mit Mandantenisolierung

Wenn Sie eine Ereignisfunktion aufrufen, für die die Mandantenisolierung aktiviert ist, müssen Sie einen Parameter angeben. `tenant-id` Dieser Parameter stellt sicher, dass Ihr Aufruf der Event-Funktion in einer Ausführungsumgebung verarbeitet wird, die für diesen bestimmten Mandanten reserviert ist.

Aufrufen von Ereignisfunktionen mit Mandantenisolierung (AWS CLI)

Synchroner Aufruf

Fügen Sie für synchrone Aufrufe den `--tenant-id` Parameter zu Ihrem Invoke-Befehl hinzu:

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    response.json
```

Asynchroner Aufruf

Geben Sie bei asynchronen Aufrufen sowohl die als auch die Parameter an: `--tenant-id --invocation-type Event`

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    --invocation-type Event \
    response.json
```

Aufrufen von Ereignisfunktionen mit Mandantenisolierung (API)

Wenn Sie die Aktion „API aufrufen“ direkt verwenden, geben Sie die Mandanten-ID mithilfe des `X-Amzn-Tenant-Id` Parameters in Ihre Anfrage ein.

Beispiel für eine API-Anfrage

```
POST /2015-03-31/functions/image-analysis/invocations HTTP/1.1
Host: lambda.us-east-1.amazonaws.com
Content-Type: application/json
Authorization: AWS4-HMAC-SHA256 Credential=...
X-Amz-Tenant-Id: blue
```

```
{
    "key1": "value1",
    "key2": "value2"
}
```

Aufrufen von Ereignisfunktionen mit Mandantenisolierung (API Gateway)

Wenn Sie API Gateway-REST-APIs verwenden, um mandantenisolierte Lambda-Ereignisfunktionen auszulösen, müssen Sie API Gateway so konfigurieren, dass die Eigenschaften der Client-Anfrage dem von Lambda erwarteten `X-Amz-Tenant-Id` Header zugeordnet werden. API Gateway verwendet die Aktion „API aufrufen“ von Lambda, bei der die Mandanten-ID über den `X-Amz-Tenant-Id` HTTP-Header übergeben werden muss. Sie können API Gateway so konfigurieren, dass dieser HTTP-Header mit einem Wert, der aus Eigenschaften der Client-Anfrage wie HTTP-Headern, Abfrageparametern oder Pfadparametern abgerufen wird, in die Lambda-Aufrufanforderung eingefügt wird. Sie müssen zuerst die Eigenschaft der Client-Anfrage zuordnen, bevor Sie den Header überschreiben können. `X-Amz-Tenant-Id`

> **Note**
>
> Sie können HTTP-APIs nicht verwenden, um mandantenisolierte Lambda-Ereignisfunktionen aufzurufen, da es nicht möglich ist, den Header zu überschreiben. `X-Amz-Tenant-Id`

Verwenden von Anforderungsheadern

Konfigurieren Sie Ihre API-Gateway-Integration so, dass ein benutzerdefinierter Header von der Client-Anfrage dem `X-Amz-Tenant-Id` Header zugeordnet wird. Das folgende Beispiel zeigt eine Client-Anfrage mit einem `x-tenant-id` Header:

```
POST /api/process HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
x-tenant-id: blue

{
    "data": "sample payload"
}
```

In Ihrer API Gateway Gateway-Methodenkonfiguration müssen Sie:

1. Aktivieren Sie den Header-Parameter für die Client-Anfrage (zum Beispiel`method.request.header.x-tenant-id`)
2. Ordnen Sie den Client-Header dem Lambda-Integrationsheader zu, indem Sie `integration.request.header.X-Amz-Tenant-Id`

Verwenden von Abfrageparametern

Auf ähnliche Weise können Sie dem `X-Amz-Tenant-Id` Header Abfrageparameter zuordnen:

```
GET /api/process?tenant-id=blue&data=sample HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
```

Konfigurieren Sie die Methode so, dass der Abfrageparameter aktiviert wird, und ordnen Sie ihn dem Integrationsheader zu.

Aufrufen von Ereignisfunktionen mit Mandantenisolierung (SDK)

Wenn Sie AWS SDKs verwenden, um von Mandanten isolierte Ereignisfunktionen aufzurufen, geben Sie die Mandanten-ID in Ihre Aufrufanforderung ein.

Python

```
import boto3
import json

lambda_client = boto3.client('lambda')

response = lambda_client.invoke(
    FunctionName='image-analysis',
    TenantId='blue',
    Payload=json.dumps({
        'key1': 'value1',
        'key2': 'value2'
    })
)

result = json.loads(response['Payload'].read())
```

Node.js

```
const AWS = require('aws-sdk');
```

```
const lambda = new AWS.Lambda();

const params = {
    FunctionName: 'image-analysis',
    TenantId: 'blue',
    Payload: JSON.stringify({
        key1: 'value1',
        key2: 'value2'
    })
};

lambda.invoke(params, (err, data) => {
    if (err) {
        console.error(err);
    } else {
        const result = JSON.parse(data.Payload);
        console.log(result);
    }
});
```

Java

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.core.SdkBytes;

public class TenantIsolationExample {

    public static void main(String[] args) {
        LambdaClient lambdaClient = LambdaClient.create();

        String payload = "{\"key1\": \"value1\", \"key2\": \"value2\"}";

        InvokeRequest request = InvokeRequest.builder()
                .functionName("image-analysis")
                .tenantId("blue")
                .payload(SdkBytes.fromUtf8String(payload))
                .build();

        InvokeResponse response = lambdaClient.invoke(request);
    }
}
```

Zugreifen auf die Mandanten-ID im Lambda-Ereignisfunktionscode

Wenn für Ihre Lambda-Ereignisfunktion die Mandantenisolierung aktiviert ist, wird die Mandanten-ID, die zum Aufrufen Ihrer Ereignisfunktion verwendet wird, in dem Kontextobjekt verfügbar gemacht, das an Ihren Event Function Handler übergeben wird. Sie können diesen Bezeichner verwenden, um mandantenspezifische Logik-, Überwachungs- und Debugging-Funktionen zu implementieren.

Themen

- Zugreifen auf die Mandanten-ID
- Allgemeine Nutzungsmuster
- Überwachung und Debugging

Zugreifen auf die Mandanten-ID

Die Mandanten-ID ist über die `tenantId` Eigenschaft des Kontextobjekts verfügbar. Beachten Sie, dass diese Eigenschaft während der Aufrufphase und nicht während der Initialisierungsphase verfügbar ist.

Python

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id
    print(f"Processing request for tenant: {tenant_id}")

    # Implement tenant-specific logic
    if tenant_id == "blue":
        return process_blue_tenant(event)
    elif tenant_id == "green":
        return process_green_tenant(event)
    else:
        return process_default_tenant(event)
```

Node.js

```
exports.handler = async (event, context) => {
    const tenantId = context.tenantId;
    console.log(`Processing request for tenant: ${tenantId}`);

    // Implement tenant-specific logic
```

```
    switch (tenantId) {
        case 'blue':
            return processBlueTenant(event);
        case 'green':
            return processGreenTenant(event);
        default:
            return processDefaultTenant(event);
    }
};
```

Java

```
public class TenantHandler implements RequestHandler<Map<String, Object>, String> {

    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        String tenantId = context.getTenantId();
        System.out.println("Processing request for tenant: " + tenantId);

        // Implement tenant-specific logic
        switch (tenantId) {
            case "blue":
                return processBlueTenant(event);
            case "green":
                return processGreenTenant(event);
            default:
                return processDefaultTenant(event);
        }
    }
}
```

Allgemeine Nutzungsmuster

Im Folgenden finden Sie gängige Methoden zur Verwendung der Mandanten-ID in Ihrem Event-Funktionscode:

Tenant-specific Konfiguration

Verwenden Sie die Mandanten-ID, um mandantenspezifische Konfigurationen oder Einstellungen zu laden:

```
def lambda_handler(event, context):
```

```
    tenant_id = context.tenant_id

    # Load tenant-specific configuration
    config = load_tenant_config(tenant_id)
    database_url = config['database_url']
    api_key = config['api_key']

    # Process with tenant-specific settings
    return process_request(event, database_url, api_key)
```

Tenant-specific Datenzugriff

Verwenden Sie die Mandanten-ID, um die Datenisolierung und Zugriffskontrolle sicherzustellen:

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Ensure data access is scoped to the tenant
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table('user_data')

    user_id = event.get('userId')

    response = table.get_item(
        Key={
            'tenant_id': tenant_id,
            'user_id': user_id
        }
    )

    return process_results(response.get('Item'), tenant_id)
```

Überwachung und Debugging

Die Mandanten-ID wird automatisch in Lambda-Logs aufgenommen, wenn Sie die JSON-Protokollierung aktiviert haben, sodass mandantenspezifische Probleme leichter überwacht und debuggt werden können. Sie können die Mandanten-ID auch für benutzerdefinierte Metriken und die Ablaufverfolgung verwenden.

Example Benutzerdefinierte Metriken mit Mandanten-ID

Das folgende Beispiel zeigt, wie Sie die Mandanten-ID verwenden, um mandantenspezifische CloudWatch Metriken zur Überwachung von Nutzungsmustern und Leistung pro Mandant zu erstellen:

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id
    cloudwatch = boto3.client('cloudwatch')

    # Record tenant-specific metrics
    cloudwatch.put_metric_data(
        Namespace='MyApp/TenantMetrics',
        MetricData=[
            {
                'MetricName': 'RequestCount',
                'Dimensions': [
                    {
                        'Name': 'TenantId',
                        'Value': tenant_id
                    }
                ],
                'Value': 1,
                'Unit': 'Count'
            }
        ]
    )

    return process_request(event, tenant_id)
```

Überwachung von Lambda-Ereignisfunktionen mit Mandantenisolierung

Sie können Ihre mandantenisolierten Lambda-Funktionen mithilfe von Amazon CloudWatch überwachen und mithilfe der Telemetrie-API auf Echtzeit-Telemetriedaten für Erweiterungen zugreifen. AWS X-Ray

Grundlegendes zur Protokollierung für den isolierten Mandantenmodus

Bei Funktionen, die Mandantenisolierung verwenden, nimmt Lambda die Mandanten-ID automatisch in Funktionsprotokolle auf, wenn Sie die JSON-Protokollierung aktiviert haben, was die Überwachung und das Debuggen mandantenspezifischer Probleme erleichtert. Lambda erstellt für jede Ausführungsumgebung einen separaten CloudWatch Protokollstream. Sie können CloudWatch Logs Insights verwenden, um Log-Streams zu finden, die zu einem bestimmten Mandanten gehören, indem Sie nach der Mandanten-ID filtern:

```
fields @logStream, @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| stats count() as logCount by @logStream
| sort @timestamp desc
```

Sie können diesen Parameter auch verwenden, um mandantenspezifische Protokolle für alle Log-Streams abzurufen:

```
fields @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| limit 1000
```

Die `tenantId` Eigenschaft ist für Plattformereignisse (wie `platform.start` und`platform.report`) und benutzerdefinierte Protokolle, die Sie in Ihrem Funktionscode drucken, enthalten, wie unten dargestellt:

```
{
    "time": "2025-10-13T19:48:06.990Z",
    "type": "platform.start",
    "record": {
        "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
        "functionArn": "arn:aws:lambda:us-east-1:xxxxxx:function:multitenant-
function-1",
        "version": "$LATEST",
        "tenantId": "BlueTenant""
```

```
    }
}
{
    "timestamp": "2025-10-13T19:48:06.992Z",
    "level": "INFO",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line1"
}
{
    "timestamp": "2025-10-13T19:48:07.022Z",
    "level": "WARN",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line2"
}
```

Problembehandlung bei der Mandantenisolierung für Lambda Event Functions

Auf dieser Seite werden häufig auftretende Probleme behandelt, die bei der Verwendung von Tenant Isolation für auftreten AWS Lambda.

InvalidParameterValueException

Fehler: Die Mandanten-ID-Konfiguration wurde nicht angegeben oder an eine Funktion übergeben, wenn die Mandantenisolierung nicht aktiviert ist.

Häufige Ursachen

Dieser Fehler tritt auf, wenn eine vom Mandanten isolierte Funktion ohne Mandanten-ID oder eine nicht vom Mandanten isolierte Funktion mit einer Mandanten-ID aufgerufen wird.

Auflösung

Fügen Sie eine Mandanten-ID hinzu, wenn für die Funktion die Mandantenisolierung aktiviert ist, oder entfernen Sie die Mandanten-ID, wenn für die Funktion die Mandantenisolierung nicht aktiviert ist.

TooManyRequestsException

Fehler: Rate überschritten

Häufige Ursachen

Zusätzlich zur Ratenbegrenzung, die auf der maximalen Anzahl gleichzeitiger Ausführungen und der Funktionsskalierungsrate basiert, begrenzt Lambda die maximale Anzahl von mandantenfähigen Ausführungsumgebungen (aktiv oder inaktiv), die gleichzeitig existieren können, auf 2.500 pro 1.000 gleichzeitige Ausführungen Ihrer Funktion.

Auflösung

Um dieses Problem zu beheben, können Sie entweder die Rate verringern, mit der Aufrufanforderungen mit eindeutigen Mandantenkennungen gestellt werden, Wiederholungsversuche mit Backoff und Jitter implementieren oder eine Erhöhung des Parallelitätslimits für Funktionen beantragen.

Von Lambda verwaltete Instances

Mit Lambda Managed Instances können Sie Lambda Event Functions auf Ihren Amazon EC2-Instances der aktuellen Generation ausführen, einschließlich Graviton4, netzwerkoptimierten Instances und anderen speziellen Rechenoptionen. Sie müssen keine Instance-Lebenszyklen, Betriebssystem- und Sprachlaufzeit-Patches, Routing, Load Balancing oder Skalierungsrichtlinien verwalten. Mit Lambda Managed Instances profitieren Sie von EC2-Preisvorteilen, einschließlich EC2-Sparplänen und Reserved Instances. Lambda Managed Instances unterstützen derzeit nur Lambda Event Functions.

Eine Liste der unterstützten Instanztypen finden Sie unter AWS Lambda Preisgestaltung Seite und wähle deine AWS-Region.

Wichtigste Funktionen

Lambda Managed Instances bietet die folgenden Funktionen:

- Wählen Sie geeignete Instanzen - Wählen Sie entsprechende Instanzen basierend auf Leistungs- und Kostenanforderungen, einschließlich Zugriff auf die neuesten CPUs wie Graviton4, konfigurierbare Speicher-CPU-Verhältnisse und Netzwerke mit hoher Bandbreite.
- Automatische Bereitstellung - AWS stellt automatisch geeignete Instanzen bereit und startet Umgebungen zur Funktionsausführung.
- Dynamische Skalierung - Instanzen werden dynamisch auf der Grundlage Ihrer Funktions-Traffic-Muster skaliert.
- Vollständig verwaltetes Erlebnis - AWS kümmert sich um Infrastrukturmanagement, Skalierung, Patching und Routing mit denselben umfangreichen Integrationen für Eventquellen, mit denen Sie vertraut sind.

Wann sollten Lambda Managed Instances verwendet werden

Ziehen Sie Lambda Managed Instances für die folgenden Anwendungsfälle in Betracht:

- Workloads mit hohem Volumen, die vorhersehbar sind - Ideal für stationäre Workloads ohne unerwartete Datenverkehrsspitzen. Lambda Managed Instances lassen sich skalieren, um standardmäßig eine Verdoppelung des Datenverkehrs innerhalb von fünf Minuten zu bewältigen.
- Performance-critical Anwendungen - Zugriff auf die neuesten CPUs, unterschiedliche Speicher-CPU-Verhältnisse und hoher Netzwerkdurchsatz

- Regulatorische Anforderungen - Granulare Governance-Anforderungen mit Kontrolle über die VPC und die Platzierung der Instanzen
- Vielzahl von Anwendungen - Event-driven - Anwendungen, media/data Verarbeitung, Webanwendungen und ältere Workloads, die auf serverlose Systeme migriert werden

Funktionsweise

Lambda Managed Instances verwendet Kapazitätsanbieter als Grundlage für die Ausführung Ihrer Funktionen:

1. Erstellen Sie einen Kapazitätsanbieter — Definieren Sie, wo Ihre Funktionen ausgeführt werden, indem Sie die VPC-Konfiguration und optional die Instance-Anforderungen sowie die Skalierungskonfiguration angeben
2. Erstellen Sie Ihre Funktion - Erstellen Sie wie gewohnt Lambda-Funktionen und hängen Sie sie an einen Kapazitätsanbieter an
3. Veröffentlichen Sie eine Funktionsversion — Funktionsversionen werden nach der Veröffentlichung auf den Instanzen des Kapazitätsanbieters aktiv

Wenn Sie eine Funktionsversion bei einem Kapazitätsanbieter veröffentlichen, startet Lambda Managed Instances in Ihrem Konto. Aus Gründen der AZ-Resilienz werden standardmäßig drei Instances gestartet und drei Ausführungsumgebungen gestartet, bevor Ihre Funktionsversion als AKTIV markiert wird. Wenn Sie eine Funktion an einen vorhandenen Kapazitätsanbieter anhängen, der bereits andere Funktionen ausführt, startet Lambda möglicherweise keine neuen Instances, wenn die verfügbaren Instances bereits über Kapazitäten für die Ausführungsumgebungen der neuen Funktion verfügen.

Modell der Parallelität

Lambda Managed Instances unterstützen mehrere gleichzeitige Aufrufe, wobei eine Ausführungsumgebung mehrere Aufrufe gleichzeitig verarbeiten kann. Dies unterscheidet sich vom Lambda-Berechnungstyp (Standard), der ein einzelnes Parallelitätsmodell bietet, bei dem eine Ausführungsumgebung maximal einen Aufruf gleichzeitig ausführen kann. Multi-concurrency sorgt für eine bessere Nutzung Ihrer zugrunde liegenden EC2-Instances und ist besonders vorteilhaft für IO-heavy Anwendungen wie Webdienste oder Batch-Jobs. Diese Änderung des Ausführungsmodells bedeutet, dass Thread-Sicherheit, Statusverwaltung und Kontextisolierung je nach Laufzeit unterschiedlich gehandhabt werden müssen.

Mietverhältnis und Isolierung

Der Lambda-Computetyp (Standard) ist mandantenfähig und nutzt die Firecracker MicroVM-Technologie, um die Isolierung zwischen Ausführungsumgebungen zu gewährleisten, die auf gemeinsam genutzten Lambda-Flotten ausgeführt werden. Lambda Managed Instances werden in Ihrem Konto ausgeführt und bieten die neueste EC2-Hardware und Preisoptionen. Verwaltete Instances verwenden Container, die auf EC2-Nitro-Instances ausgeführt werden, um die Isolierung zu gewährleisten, und nicht Firecracker. Kapazitätsanbieter dienen als Sicherheitsgrenze für Lambda-Funktionen. Funktionen werden in Containern innerhalb von Instanzen ausgeführt.

Grundlegendes zu verwalteten Instanzen

Die Funktionen von Lambda Managed Instances werden ausgeführt auf Von EC2 verwaltete Instanzen in Ihrem Konto. Diese Instances werden vollständig von Lambda verwaltet, was bedeutet, dass Sie im Vergleich zu Standard-EC2-Instances nur über eingeschränkte Berechtigungen für sie verfügen. Sie können Lambda Managed Instances in Ihrem Konto wie folgt identifizieren:

- Das Vorhandensein von `Operator` Feld in EC2 `DescribeInstances` Ausgang
- Die `aws:lambda:capacity-provider` Tag auf der Instanz

Sie können Standard-EC2-Operationen nicht direkt auf diesen Instances ausführen, z. B. sie manuell beenden. Um verwaltete Instances zu löschen, löschen Sie den zugehörigen Kapazitätsanbieter. Lambda beendet dann die Instances im Rahmen des Löschvorgangs des Kapazitätsanbieters.

Verwaltete Instances sind standardmäßig in Ihren EC2-Konsolenansichten und API-Listenvorgängen verborgen. Sie können die Sichtbarkeit anpassen, indem Sie Einstellung für die Sichtbarkeit verwalteter Ressourcen. Verwaltete Instanzen bleiben in Ihrem Unternehmen voll funktionsfähig und abrechnungsfähig AWS Konto.

Preisgestaltung

Lambda Managed Instances verwendet EC2-based Preisgestaltung mit einer Verwaltungsgebühr von 15% zusätzlich zu den Kosten für EC2-Instances. Dieses Preismodell unterstützt EC2-Sparpläne, Reserved Instances und alle anderen Preisnachlässe, die für Ihre EC2-Nutzung gelten. Weitere Informationen finden Sie auf der Preisseite: https://aws.amazon.com/lambda/pricing/

Um die Kosten für die Ausführung Ihrer Funktionen auf Lambda Default, Lambda Managed Instances und selbstverwaltetem Amazon EC2 zu vergleichen, verwenden Sie Preisrechner für Lambda Managed Instances.

wichtig: EC2-Preisnachlässe gelten nur für die zugrunde liegende EC2-Berechnung, nicht für die Verwaltungsgebühr.

Wie unterscheidet sich Lambda Managed Instances vom Lambda-Berechnungstyp (Standard)

Lambda Managed Instances verändert die Art und Weise, wie Lambda Anfragen verarbeitet, im Vergleich zu Lambda (Standard).

Die wichtigsten Unterschiede:

	Lambda (Standard)	Von Lambda verwaltete Instances
Modell der Parallelität	Einzelnes Parallelitätsmodell, bei dem eine Ausführungsumgebung maximal einen Aufruf gleichzeitig unterstützen kann	Multi-concurrent Aufrufe, bei denen eine Ausführungsumgebung mehrere Aufrufe gleichzeitig verarbeiten kann, wodurch der Durchsatz insbesondere für erhöht wird IO-heavy Anwendungen
Mietverhältnis und Isolierung	Multi-tenant, verwendet die Firecracker MicroVM-Technologie, um die Isolierung zwischen Ausführungsumgebungen zu gewährleisten, die auf gemeinsam genutzten Lambda-Flotten ausgeführt werden	Führen Sie es in Ihrem Konto aus und verwenden Sie EC2 Nitro, um die Isolierung zu gewährleisten. Kapazitätsanbieter dienen als Sicherheitsgrenze, wobei Funktionen in Containern innerhalb von Instances ausgeführt werden
Preismodell	Per-request Dauer, Preisgestaltung	Instance-based Preisgestaltung mit EC2-Preismodellen, einschließlich On-Demand und Reserved Instances sowie Sparoptionen wie Compute Savings Plans
Skalierungsverhalten	Skaliert, wenn es keine freie Ausführungsumgebung gibt, um einen eingehenden Aufruf zu	Skaliert asynchron nur auf Grundlage der CPU-Ressourcenauslastung, ohne Kaltstarts. Skaliert auf ein Minimum an

	Lambda (Standard)	Von Lambda verwaltete Instances
	verarbeiten (Kaltstart). Skaliert ohne Verkehr auf Null	Ausführungsumgebungen, die ohne Datenverkehr konfiguriert sind
Am besten geeignet für	Funktionen mit hohem Datenaufkommen, die auch mal einen Kaltstart verkraften können, oder Anwendungen ohne Dauerlast, die von einer Skalierung auf Null profitieren	Hochvolumiger, vorhersehbarer Datenverkehr funktioniert, wenn Sie die Flexibilität, die Preispläne und die Hardwareoptionen von EC2 nutzen möchten

Nächste Schritte

- Erfahren Sie Kernkonzepte für Lambda Managed Instances
- Beginnen Sie mit Lambda Managed Instances
- Erfahren Sie Kapazitätsanbieter für Lambda Managed Instances
- Verstehen Skalierung für von Lambda verwaltete Instances
- Erfahren Sie mehr über Bereitstellung und Aktualisierung der Funktionen von Lambda Managed Instances
- Überprüfung unterstützte Laufzeiten
- Lesen Sie die laufzeitspezifischen Anleitungen für Java, Node.js, und Python
- Konfiguriere VPC-Konnektivität für Ihre Kapazitätsanbieter
- Verstehen Sicherheit und Berechtigungen für Lambda Managed Instances
- Überprüfung Kontingente und Grenzwerte
- Beheben Sie Probleme mit Lambda Managed Instances

Erste Schritte mit Lambda Managed Instances

Eine Lambda Managed Instance-Funktion erstellen (Konsole)

Sie können die Lambda-Konsole verwenden, um eine Managed Instance-Funktion zu erstellen, die auf Amazon EC2 EC2-Instances ausgeführt wird, die von einem Kapazitätsanbieter verwaltet werden.

Wichtig: Bevor Sie eine Managed Instance-Funktion erstellen, müssen Sie zunächst einen Kapazitätsanbieter erstellen. Für diese Lambda-Ereignisfunktionen ist ein Kapazitätsanbieter erforderlich, um die Amazon EC2 EC2-Infrastruktur zu definieren, auf der Ihre Funktionen ausgeführt werden.

So erstellen Sie eine Lambda Managed Instance-Funktion (Konsole)

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie im linken Navigationsbereich Capacity Providers aus.
3. Wählen Sie Kapazitätsanbieter erstellen aus.
4. Geben Sie im Abschnitt Einstellungen für den Kapazitätsanbieter einen Namen für Ihren Kapazitätsanbieter ein.
5. Wählen Sie VPC und Berechtigungen für Ihren Kapazitätsanbieter aus. Sie können entweder ein vorhandenes verwenden oder ein neues erstellen. Informationen zum Erstellen der erforderlichen Operatorrolle finden Sie unter Lambda-Operatorrolle für Lambda Managed Instances.
6. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
7. Definieren Sie Ihre Instance-Anforderungen, indem Sie die Prozessorarchitektur und die Instance-Typen auswählen.
8. Geben Sie unter Auto Scaling die maximale Anzahl von EC2-vCPUs für Ihren Kapazitätsanbieter an. Sie können auch den Modus Manuelle Instanzskalierung wählen, um Ihren eigenen Skalierungswert für eine präzise Steuerung festzulegen.
9. Wählen Sie Create Capacity Provider aus, um einen neuen zu erstellen.
10. Wählen Sie Create function.
11. Wählen Sie Verfassen von Grund auf aus.
12. Geben Sie im Bereich Grundinformationen einen Funktionsnamen ein.
13. Wählen Sie für Runtime eine der unterstützten Runtimes aus.
14. Wählen Sie die Architektur für Ihre Funktion (dieselbe, die Sie für den Kapazitätsanbieter ausgewählt haben). Standardmäßig x86_64.
15. Stellen Sie unter Berechtigungen sicher, dass Sie über die Berechtigungen für die gewählte Ausführungsrolle verfügen. Andernfalls können Sie eine neue Rolle erstellen.
16. Wählen Sie unter Zusätzliche Konfigurationen den Compute-Typ als Lambda Managed Instances aus.
17. Der Kapazitätsanbieter-ARN des Kapazitätsanbieters, den Sie in den vorherigen Schritten erstellt haben, sollte vorab ausgewählt sein.

18. Wählen Sie Speichergröße und Verhältnis Arbeitsspeicher (GiB) pro vCPU aus.
19. Wählen Sie Funktion erstellen.

Ihre Lambda Managed Instance-Funktion wird erstellt und stellt Kapazität auf Ihrem angegebenen Kapazitätsanbieter bereit. Die Erstellung von Funktionen dauert in der Regel mehrere Minuten. Sobald der Vorgang abgeschlossen ist, können Sie Ihren Funktionscode bearbeiten und Ihren ersten Test ausführen.

Eine Lambda Managed Instance-Funktion erstellen (AWS CLI)

Voraussetzungen

Stellen Sie vor dem Beginn sicher, dass Sie über das Folgende verfügen:

- AWS CLI — Installieren und konfigurieren Sie die AWS CLI. Weitere Informationen finden Sie unter Installation oder Aktualisierung der neuesten Version der AWS CLI.
- IAM-Berechtigungen — Ihr IAM-Benutzer oder Ihre IAM-Rolle muss über Berechtigungen verfügen, um Lambda-Funktionen und Kapazitätsanbieter zu erstellen und IAM-Rollen zu übergeben. Beachten Sie, dass Sie auch angeben müssen, `iam:CreateServiceLinkedRole` ob Sie zum ersten Mal einen Kapazitätsanbieter für das Konto erstellen oder ob die Service Linked Role (SLR) gelöscht wurde.

Schritt 1: Erstellen Sie die erforderlichen IAM-Rollen

Lambda Managed Instances erfordern zwei IAM-Rollen: eine Ausführungsrolle für Ihre Funktion und eine Operator-Rolle für den Kapazitätsanbieter. Die Operator-Rolle ermöglicht es Lambda, Amazon EC2 EC2-Instances in Ihrem Namen zu starten, zu beenden und zu überwachen. Die Funktionsausführungsrolle gewährt der Funktion Berechtigungen für den Zugriff auf andere AWS Dienste und Ressourcen.

So erstellen Sie die Lambda-Ausführungsrolle

1. Erstellen Sie ein Dokument mit Vertrauensrichtlinien, das es Lambda ermöglicht, die Rolle zu übernehmen:

```
cat > lambda-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

2. Erstellen Sie die Ausführungsrolle:

```
aws iam create-role \
  --role-name MyLambdaExecutionRole \
  --assume-role-policy-document file://lambda-trust-policy.json
```

3. Hängen Sie die grundlegende Ausführungsrichtlinie an:

```
aws iam attach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

Um die Rolle des Kapazitätsanbieters als Betreiber zu erstellen

1. Erstellen Sie ein Dokument mit Vertrauensrichtlinien, das es Lambda ermöglicht, die Operatorrolle zu übernehmen:

```
cat > operator-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
EOF
```

2. Erstellen Sie die Operatorrolle:

```
aws iam create-role \
  --role-name MyCapacityProviderOperatorRole \
  --assume-role-policy-document file://operator-trust-policy.json
```

3. Hängen Sie die erforderliche EC2-Berechtigungsrichtlinie an:

```
aws iam attach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator
```

Schritt 2: VPC-Ressourcen einrichten

Lambda Managed Instances werden in Ihrer VPC ausgeführt und benötigen ein Subnetz und eine Sicherheitsgruppe.

Um VPC-Ressourcen zu erstellen

1. Erstellen Sie eine VPC:

```
VPC_ID=$(aws ec2 create-vpc \
  --cidr-block 10.0.0.0/16 \
  --query 'Vpc.VpcId' \
  --output text)
```

2. Erstellen Sie ein Subnetz:

```
SUBNET_ID=$(aws ec2 create-subnet \
  --vpc-id $VPC_ID \
  --cidr-block 10.0.1.0/24 \
  --query 'Subnet.SubnetId' \
  --output text)
```

3. Erstellen Sie eine Sicherheitsgruppe:

```
SECURITY_GROUP_ID=$(aws ec2 create-security-group \
  --group-name my-capacity-provider-sg \
  --description "Security group for Lambda Managed Instances" \
```

```
  --vpc-id $VPC_ID \
  --query 'GroupId' \
  --output text)
```

Hinweis: Ihre Lambda Managed Instance-Funktionen erfordern eine VPC-Konfiguration, um auf Ressourcen außerhalb der VPC zuzugreifen und Telemetriedaten an Logs und zu übertragen. CloudWatch X-Ray Einzelheiten zur Konfiguration finden Sie unter Networking for Lambda Managed Instances.

Schritt 3: Erstellen eines Kapazitätsanbieters

Ein Kapazitätsanbieter verwaltet die EC2-Instances, auf denen Ihre Lambda-Funktionen ausgeführt werden.

Um einen Kapazitätsanbieter zu erstellen

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)

aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=[$SUBNET_ID],SecurityGroupIds=[$SECURITY_GROUP_ID] \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::${ACCOUNT_ID}:role/
MyCapacityProviderOperatorRole \
  --instance-requirements Architectures=[x86_64] \
  --capacity-provider-scaling-config MaxVCpuCount=30
```

Dieser Befehl erstellt einen Kapazitätsanbieter mit der folgenden Konfiguration:

- VPC-Konfiguration — Gibt das Subnetz und die Sicherheitsgruppe für die EC2-Instances an
- Berechtigungen — Definiert die IAM-Rolle, die Lambda zur Verwaltung von EC2-Instances verwendet
- Instanzanforderungen — Spezifiziert die x86_64-Architektur
- Skalierungskonfiguration — Legt ein Maximum von 30 vCPUs für den Kapazitätsanbieter fest

Schritt 4: Erstellen Sie eine Lambda-Funktion mit Inline-Code

Um eine Funktion mit Inline-Code zu erstellen

1. Erstellen Sie zunächst eine einfache Python-Funktion und verpacken Sie sie inline:

```
# Create a temporary directory for the function code
mkdir -p /tmp/my-lambda-function
cd /tmp/my-lambda-function

# Create a simple Python handler
cat > lambda_function.py << 'EOF'
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps({
            'message': 'Hello from Lambda Managed Instances!',
            'event': event
        })
    }
EOF

# Create a ZIP file
zip function.zip lambda_function.py
```

2. Erstellen Sie die Lambda-Funktion mithilfe der Inline-ZIP-Datei:

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=$(aws configure get region)

aws lambda create-function \
  --function-name my-managed-instance-function \
  --package-type Zip \
  --runtime python3.13 \
  --handler lambda_function.lambda_handler \
  --zip-file fileb:///tmp/my-lambda-function/function.zip \
  --role arn:aws:iam::${ACCOUNT_ID}:role/MyLambdaExecutionRole \
  --architectures x86_64 \
  --memory-size 2048 \
  --ephemeral-storage Size=512 \
  --capacity-provider-config
 LambdaManagedInstancesCapacityProviderConfig={CapacityProviderArn=arn:aws:lambda:
${REGION}:${ACCOUNT_ID}:capacity-provider:my-capacity-provider}
```

Die Funktion wird erstellt mit:

- Laufzeit — Python 3.13
- Handler — Die `lambda_handler` Funktion in `lambda_function.py`
- Speicher — 2048 MB
- Kurzlebiger Speicher — 512 MB
- Kapazitätsanbieter — Links zu dem Kapazitätsanbieter, den Sie erstellt haben

Schritt 5: Veröffentlichen Sie eine Funktionsversion

Um Ihre Funktion auf Lambda Managed Instances auszuführen, müssen Sie eine Version veröffentlichen.

Um eine Funktionsversion zu veröffentlichen

```
aws lambda publish-version \
  --function-name my-managed-instance-function
```

Dieser Befehl veröffentlicht Version 1 Ihrer Funktion und stellt sie für den Kapazitätsanbieter bereit.

Schritt 6: Rufen Sie Ihre Funktion auf

Nach der Veröffentlichung können Sie Ihre Funktion aufrufen.

Um Ihre Funktion aufzurufen

```
aws lambda invoke \
  --function-name my-managed-instance-function:1 \
  --payload '{"name": "World"}' \
  response.json

# View the response
cat response.json
```

Die Funktion wird auf den von Ihrem Kapazitätsanbieter verwalteten EC2-Instances ausgeführt und gibt eine Antwort zurück.

Bereinigen

Um Gebühren zu vermeiden, löschen Sie die Ressourcen, die Sie erstellt haben:

1. Löschen Sie die Funktion:

```
aws lambda delete-function --function-name my-managed-instance-function
```

2. Löschen Sie den Kapazitätsanbieter:

```
aws lambda delete-capacity-provider --capacity-provider-name my-capacity-provider
```

3. Löschen Sie die VPC-Ressourcen:

```
aws ec2 delete-security-group --group-id $SECURITY_GROUP_ID
aws ec2 delete-subnet --subnet-id $SUBNET_ID
aws ec2 delete-vpc --vpc-id $VPC_ID
```

4. Löschen Sie die IAM-Rollen:

```
aws iam detach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
aws iam detach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator

aws iam delete-role --role-name MyLambdaExecutionRole
aws iam delete-role --role-name MyCapacityProviderOperatorRole
```

Schlüsselkonzepte

Lambda Managed Instances führt mehrere Kernkonzepte ein, die sich von herkömmlichen Lambda-Funktionen unterscheiden. Das Verständnis dieser Konzepte ist für die effektive Bereitstellung und Verwaltung Ihrer Lambda-Event-Funktionen auf der EC2-Infrastruktur unerlässlich.

Anbieter von Kapazitäten bilden die Grundlage von Lambda Managed Instances. Ein Kapazitätsanbieter definiert die Recheninfrastruktur, in der Ihre Funktionen ausgeführt werden, einschließlich VPC-Konfiguration, Instance-Anforderungen und Skalierungsrichtlinien. Kapazitätsanbieter dienen auch als Sicherheitsgrenze für Ihre Funktionen. Das bedeutet, dass alle Funktionen, die demselben Kapazitätsanbieter zugewiesen sind, gegenseitig vertrauenswürdig sein müssen.

Verhalten bei der Skalierung unterscheidet sich erheblich von herkömmlichen Lambda-Funktionen. Anstatt bei Bedarf zu skalieren, wenn Aufrufe eintreffen, skalieren Managed Instances asynchron auf

der Grundlage der CPU-Ressourcenauslastung. Dieser Ansatz vermeidet Kaltstarts, erfordert jedoch eine Planung für das Wachstum des Datenverkehrs. Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, da Lambda die Kapazität entsprechend der Nachfrage hochskaliert.

Sicherheit und Berechtigungen erfordern eine sorgfältige Abwägung. Sie benötigen Betreiberrollenberechtigungen, damit Lambda die EC2-Ressourcen Ihrer Kapazitätsanbieter verwalten kann. Darüber hinaus benötigen Benutzer die `lambda:PassCapacityProvider` die Erlaubnis, den Kapazitätsanbietern Funktionen zuzuweisen. Sie dient als Sicherheitsschleuse, um zu kontrollieren, welche Funktionen auf einer bestimmten Infrastruktur ausgeführt werden können.

Multi-concurrent Ausführung ist ein wesentliches Merkmal von Managed Instances. Jede Ausführungsumgebung kann mehrere Aufrufe gleichzeitig verarbeiten, wodurch die Ressourcenauslastung für maximiert wird IO-heavy Anwendungen. Dies unterscheidet sich von herkömmlichem Lambda, bei dem jede Umgebung jeweils eine Anfrage verarbeitet. Bei diesem Ausführungsmodell müssen je nach Laufzeit die Thread-Sicherheit, die Statusverwaltung und die Kontextisolierung berücksichtigt werden.

Einzelheiten zur Veröffentlichung von Funktionsversionen bei Ihrem Kapazitätsanbieter finden Sie unter the section called “Veröffentlichung der Version”.

Die folgenden Abschnitte enthalten detaillierte Informationen zu den einzelnen Kernkonzepten.

Kapazitätsanbieter

Ein Kapazitätsanbieter ist die Grundlage für den Betrieb von Lambda Managed Instances. Er dient als Sicherheitsgrenze für Ihre Lambda Event Functions und definiert die Rechenressourcen, die Lambda in Ihrem Namen bereitstellt und verwaltet.

Wenn Sie einen Kapazitätsanbieter erstellen, geben Sie Folgendes an:

- VPC-Konfiguration — Die Subnetze und Sicherheitsgruppen, in denen Instances ausgeführt werden
- - Berechtigungen - IAM-Rollen für Lambda zur Verwaltung von EC2-Ressourcen
- Anforderungen an die Instanz (optional) — Architektur und Instanztyp Präferenzen
- Konfiguration skalieren (optional) — Wie Lambda Ihre Instances skaliert

Kapazitätsanbieter als Sicherheitsgrenze verstehen

Kapazitätsanbieter dienen als Sicherheitsgrenze für Lambda-Funktionen in Ihrer VPC und ersetzen Firecracker-based Isolierung. Funktionen werden in Containern innerhalb von Instanzen ausgeführt, aber Container bieten im Gegensatz zu Firecracker MicroVMs keine starke Sicherheitsisolierung zwischen Funktionen.

Wichtige Sicherheitskonzepte:

- Kapazitätsanbieter: Die Sicherheitsgrenze, die Vertrauensstufen für Lambda-Funktionen definiert
- Isolierung von Containern: Container sind KEIN Sicherheitsanbieter — verlassen Sie sich nicht auf sie, wenn es um die Sicherheit zwischen nicht vertrauenswürdigen Workloads geht
- Trennung von Vertrauen: Trennen Sie Workloads, denen kein gegenseitiges Vertrauen entgegengebracht wird, indem Sie unterschiedliche Kapazitätsanbieter verwenden

Erstellen eines Kapazitätsanbieters

Sie können einen Kapazitätsanbieter mithilfe der AWS CLI, AWS Managementkonsole oder AWS SDKs.

Verwenden AWS CLI:

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config
 SubnetIds=subnet-12345,subnet-67890,subnet-11111,SecurityGroupIds=sg-12345 \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/
MyOperatorRole \
  --instance-requirements Architectures=x86_64 \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Erforderliche Parameter

CapacityProviderName

- Ein eindeutiger Name für Ihren Kapazitätsanbieter
- Muss innerhalb Ihrer eindeutig sein AWS Konto

VpcConfig

- SubnetIds (erforderlich): Mindestens ein Subnetz, maximal 16. Verwenden Sie Subnetze in mehreren Availability Zones, um die Stabilität zu gewährleisten
- SecurityGroupIds (optional): Sicherheitsgruppen für Ihre Instances. Standardmäßig wird die VPC-Standardsicherheitsgruppe verwendet, falls nicht angegeben

PermissionsConfig

- CapacityProviderOperatorRoleArn (erforderlich): IAM-Rolle, die es Lambda ermöglicht, EC2-Ressourcen in Ihrem Kapazitätsanbieter zu verwalten

Optionale Parameter

InstanceRequirements

Geben Sie die Architektur an und Instanztypen für Ihren Kapazitätsanbieter:

- Architekturen: Wählen `x86_64` oder `arm64`. Die Standardeinstellung ist `x86_64`
- AllowedInstanceTypes: Geben Sie zulässige Instanztypen oder ganze Instanzfamilien an. Sie können einen Platzhalter verwenden (`*`), um die Instance-Größen innerhalb einer bestimmten Instance-Typfamilie abzugleichen. Der Platzhalter gilt nur für den Größenbereich (nach dem Punkt). Beispiel:
  - `m5.8xlarge` — Entspricht dem exakten Instanztyp
  - `c6i.*` — Entspricht allen Größen im `C6i` Familie (`c6i.xlarge`, `c6i.2xlarge`, `c6i.4xlarge`, und so weiter)

  Platzhalter für Instanzfamilien oder Generationen werden nicht unterstützt. Zum Beispiel `c5*.*`, `r*`, und `*3*` sind keine gültigen Muster für Lambda Managed Instances.
- ExcludedInstanceTypes: Geben Sie die auszuschließenden Instanztypen an. Verwenden Sie dabei dasselbe Format wie AllowedInstanceTypes. Sie können nur eines von angeben AllowedInstanceTypes oder ExcludedInstanceTypes.

Standardmäßig wählt Lambda die optimalen Instance-Typen für Ihre Arbeitslast aus. Wir empfehlen, Lambda Managed Instances die Instanztypen für Sie auswählen zu lassen, da eine Beschränkung der Anzahl möglicher Instanztypen zu einer geringeren Verfügbarkeit führen kann.

CapacityProviderScalingConfig

Konfigurieren Sie, wie Lambda Ihre Instances skaliert:

- ScalingMode: Eingestellt auf `Auto` für automatische Skalierung oder `Manual` zur manuellen Steuerung. Die Standardeinstellung ist `Auto`
- MaxVCpuCount: Maximale Anzahl von vCPUs für den Kapazitätsanbieter. Die Standardeinstellung ist 400.
- ScalingPolicies: Definieren Sie Ziel-Tracking-Skalierungsrichtlinien für die CPU- und Speicherauslastung

KmsKeyArn

Geben Sie eine an AWS KMS Schlüssel für die EBS-Verschlüsselung. Die Standardeinstellung ist AWS verwalteter Schlüssel, falls nicht angegeben.

Tags

Fügen Sie Stichwörter hinzu, um Ihre Kapazitätsanbieter zu organisieren und zu verwalten.

PropagateTags

Konfigurieren Sie die Tag-Propagierung so, dass Tags automatisch auf verwaltete Ressourcen (Amazon EC2-Instances, Amazon EBS-Volumes und ENIs) angewendet werden, die vom Kapazitätsanbieter gestartet wurden. Verwenden Sie die Tag-Propagierung, um Kostenzuweisung, SCPs und Compliance-Anforderungen durchzusetzen. Weitere Informationen finden Sie unter Propagierung von Stichwörtern.

Verwaltung von Kapazitätsanbietern

Aktualisieren eines Kapazitätsanbieters

Sie können bestimmte Eigenschaften eines Kapazitätsanbieters mithilfe der `UpdateCapacityProvider` API.

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Löschen eines Kapazitätsanbieters

Sie können einen Kapazitätsanbieter löschen, wenn er nicht mehr benötigt wird, indem Sie den `DeleteCapacityProvider` API.

```
aws lambda delete-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Hinweis: Sie können keinen Kapazitätsanbieter löschen, dem Funktionsversionen zugeordnet sind.

Details zum Kapazitätsanbieter anzeigen

Rufen Sie Informationen über einen Kapazitätsanbieter mithilfe der `GetCapacityProvider` API.

```
aws lambda get-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Staaten des Kapazitätsanbieters

Ein Kapazitätsanbieter kann sich in einem der folgenden Bundesstaaten befinden:

- Ausstehend: Der Kapazitätsanbieter wird gerade erstellt
- Aktiv: Der Kapazitätsanbieter ist einsatzbereit
- Fehlgeschlagen: Die Erstellung des Kapazitätsanbieters ist fehlgeschlagen
- Löschen: Der Kapazitätsanbieter wird gelöscht

Kontingente

- Anbieter für maximale Kapazität pro Konto: 1.000
- Maximale Funktionsversionen pro Kapazitätsanbieter: 100 (kann nicht erhöht werden)

Bewährte Methoden

1. Getrennt nach Vertrauensstufe: Erstellen Sie verschiedene Kapazitätsanbieter für Workloads mit unterschiedlichen Sicherheitsanforderungen
2. Verwenden Sie aussagekräftige Namen: Benennen Sie die Kapazitätsanbieter, um deren Verwendungszweck und Vertrauensstufe deutlich zu kennzeichnen (z. B. `production-trusted`, `dev-sandbox`).
3. Verwenden Sie mehrere Availability Zones: Geben Sie Subnetze über mehrere AZs hinweg an, um eine hohe Verfügbarkeit zu gewährleisten

4. Lassen Sie Lambda die Instanztypen auswählen: Erlauben Sie Lambda, die für die Verfügbarkeit optimalen Instance-Typen auszuwählen, sofern Sie keine spezifischen Hardwareanforderungen haben
5. Überwachen Sie die Nutzung: Verwenden AWS CloudTrail zur Überwachung der Zuweisungen und Zugriffsmuster der Kapazitätsanbieter

Nächste Schritte

- Erfahren Sie Skalierung von Lambda Managed Instances
- Verstehen Sicherheit und Berechtigungen für Lambda Managed Instances
- Konfigurieren VPC-Konnektivität für Ihre Kapazitätsanbieter
- Lesen Sie die laufzeitspezifischen Anleitungen für Java, Node.js, und Python

Verbreitung von Tags

Mit der Tag-Propagierung können Sie eine Reihe von Tags in Ihrer Kapazitätsanbieter-Konfiguration angeben, und Lambda wendet diese Tags automatisch auf alle verwalteten Ressourcen an, die es erstellt, einschließlich Amazon EC2 EC2-Instances, Amazon EBS-Volumes und ENIs. Dadurch wird eine konsistente Kennzeichnung für Kostenzuweisung, Service Control Policies (SCPs) und Compliance-Anforderungen gewährleistet, ohne dass manuelles Eingreifen oder benutzerdefinierte Automatisierung erforderlich sind.

Konfiguration der Tag-Verbreitung

Geben Sie die `PropagateTags` Einstellung an, wenn Sie einen Kapazitätsanbieter mithilfe der `UpdateCapacityProvider` APIs oder erstellen `CreateCapacityProvider` oder aktualisieren.

PropagateTags Parameter:

- Mode (erforderlich) — Der Tag-Propagationsmodus:
  - `Explicit`— Die in angegebenen Tags werden `ExplicitTags` an verwaltete Ressourcen weitergegeben.
  - `None`— Deaktiviert die Tag-Propagierung. Auf verwaltete Ressourcen werden keine benutzerdefinierten Tags angewendet, mit Ausnahme von Systemtags, die standardmäßig angewendet werden.

- ExplicitTags(erforderlich, wenn Mode aktiviert ist`Explicit`) — Eine Zuordnung von Schlüssel-Wert-Paaren, die auf verwaltete Ressourcen angewendet werden sollen. Sie können bis zu 40 Tags angeben.

> **Important**
>
> Die Tag-Weitergabe gilt nur für neue verwaltete Ressourcen, die nach der Anwendung der Konfiguration bereitgestellt werden. Bestehende Ressourcen werden nicht rückwirkend gekennzeichnet.

Beispiele

Erstellen Sie einen Kapazitätsanbieter mit Tag-Propagation (AWS CLI):

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=subnet-12345,subnet-67890,SecurityGroupIds=sg-12345 \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/
MyOperatorRole \
  --propagate-tags '{"Mode": "Explicit", "ExplicitTags": {"CostCenter": "12345",
 "Environment": "Production"}}'
```

Aktualisieren Sie einen vorhandenen Kapazitätsanbieter, um die Tag-Weitergabe zu aktivieren:

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --propagate-tags '{"Mode": "Explicit", "ExplicitTags": {"CostCenter": "12345",
 "Environment": "Production"}}'
```

Tag-Propagierung deaktivieren:

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --propagate-tags '{"Mode": "None"}'
```

Verhalten bei der Tag-Propagierung

- Änderungen an der `PropagateTags` Konfiguration wirken sich nur auf Ressourcen aus, die nach dem Update bereitgestellt wurden. Zuvor gestartete Ressourcen behalten ihre ursprünglichen Tags.
- Tags zur Tag-Propagierung unterscheiden sich von den `Tags` Parametern auf dem Kapazitätsanbieter selbst. Kapazitätsanbieter-Tags identifizieren die Kapazitätsanbieter-Ressource; weitergegebene Tags werden auf die verwalteten Ressourcen angewendet, die der Kapazitätsanbieter startet.
- Verbreitete Tags werden auf die Grenzwerte der AWS Ressourcen-Tags für die Zielressourcen angerechnet.

Skalierung von mit Lambda verwalteten Instances

Lambda Managed Instances skaliert nicht, wenn Aufrufe eintreffen, und unterstützt keine Kaltstarts. Stattdessen skaliert es asynchron mithilfe von Signalen zum Ressourcenverbrauch. Managed Instances werden derzeit auf der Grundlage der CPU-Ressourcenauslastung und der Auslastung mehrerer Parallelitäten skaliert.

Die wichtigsten Unterschiede:

- Lambda (Standard): Skaliert, wenn es keine freie Ausführungsumgebung gibt, um einen eingehenden Aufruf zu verarbeiten (Kaltstart)
- Lambda Managed Instances: Skaliert asynchron auf der Grundlage der CPU-Ressourcenauslastung und der Auslastung der Ausführungsumgebungen mit mehreren Parallelitäten

Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, da Lambda Instances und Ausführungsumgebungen entsprechend der Nachfrage hochskaliert.

Der Lebenszyklus der Skalierung

Lambda Managed Instances verwendet eine verteilte Architektur zur Verwaltung der Skalierung:

Komponenten:

- Verwaltete Instanzen — Führen Sie sie in Ihrem Konto in den von Ihnen bereitgestellten Subnetzen aus
- Router und Scaler — Gemeinsam genutzte Lambda-Komponenten, die Aufrufe weiterleiten und die Skalierung verwalten
- Lambda Agent — Wird auf jeder verwalteten Instanz ausgeführt, um den Lebenszyklus der Ausführungsumgebung zu verwalten und den Ressourcenverbrauch zu überwachen

So funktioniert es:

1. Wenn Sie eine Funktionsversion bei einem Kapazitätsanbieter veröffentlichen, startet Lambda Managed Instances in Ihrem Konto. Aus Gründen der AZ-Resilienz werden standardmäßig drei Ausführungsumgebungen gestartet und drei Ausführungsumgebungen gestartet, bevor Ihre Funktionsversion als AKTIV markiert wird.
2. Jede verwaltete Instanz kann Ausführungsumgebungen für mehrere Funktionen ausführen, die demselben Kapazitätsanbieter zugeordnet sind.
3. Wenn der Datenverkehr in Ihre Anwendung fließt, verbrauchen die Ausführungsumgebungen Ressourcen. Der Lambda Agent benachrichtigt den Scaler, der entscheidet, ob neue Ausführungsumgebungen oder Managed Instances skaliert werden sollen.
4. Wenn der Router versucht, einen Aufruf an eine Ausführungsumgebung mit hohem Ressourcenverbrauch zu senden, benachrichtigt ihn der Lambda-Agent auf dieser Instanz, dass er es auf einer anderen Instanz erneut versuchen soll.
5. Wenn der Datenverkehr abnimmt, benachrichtigt der Lambda Agent Scaler, der dann entscheidet, die Ausführungsumgebungen herunterzufahren und in Managed Instances zu skalieren.

Anpassung des Skalierungsverhaltens

Sie können das Skalierungsverhalten von Managed Instances mithilfe von fünf Steuerelementen anpassen:

Steuerungen auf Funktionsebene

1. Funktionsspeicher und vCPUs

Wählen Sie die Speichergröße und die vCPU-Zuweisung für Ihre Funktion. Die kleinste unterstützte Funktionsgröße ist 2 GB und 1 vCPU.

Überlegungen:

- Wählen Sie eine Speicher- und vCPU-Einstellung, die die mehrfache Ausführung Ihrer Funktion unterstützt
- Sie können keine Funktion mit weniger als 1 vCPU konfigurieren, da Funktionen, die auf verwalteten Instanzen ausgeführt werden, mehrere gleichzeitige Arbeitslasten unterstützen sollten
- Sie können nicht weniger als 2 GB wählen, da dies dem Verhältnis von Arbeitsspeicher zu vCPU von 2 zu 1 von c-Instances entspricht, die das niedrigste Verhältnis aufweisen
- Für Python-Anwendungen müssen Sie möglicherweise ein höheres Verhältnis von Arbeitsspeicher zu vCPUs wählen, z. B. 4 zu 1 oder 8 zu 1, da Python mehrere Parallelitäten verarbeitet
- Wenn Sie CPU-intensive Operationen ausführen oder wenig I/O ausführen, sollten Sie mehr als eine vCPU wählen

2. Maximale Parallelität

Legen Sie die maximale Parallelität pro Ausführungsumgebung fest.

Standardverhalten: Lambda wählt sinnvolle Standardwerte, die den Ressourcenverbrauch und den Durchsatz ausgleichen und für eine Vielzahl von Anwendungen funktionieren.

Richtlinien für Anpassungen:

- Erhöhen Sie die Parallelität: Wenn Ihre Funktionsaufrufe sehr wenig CPU beanspruchen, können Sie die maximale Parallelität auf maximal 64 pro vCPU erhöhen
- Verringern Sie die Parallelität: Wenn Ihre Anwendung viel Arbeitsspeicher und nur sehr wenig CPU verbraucht, können Sie Ihre maximale Parallelität reduzieren

Wichtig: Da Lambda Managed Instances für Anwendungen mit mehreren gleichzeitigen Vorgängen konzipiert sind, kann es in Ausführungsumgebungen mit sehr geringer Parallelität bei der Skalierung zu Drosselungen kommen. Wenn Aufrufe in einer Ausführungsumgebung ankommen, die ihre Parallelitätsgrenze erreicht hat, leitet Lambda diese Aufrufe an eine andere Stelle weiter und skaliert neue Ausführungsumgebungen, um die Last zu bewältigen. Um zu ermitteln, welche Ressourceneinschränkung die Drosselung verursacht, überwachen Sie die Metriken für den Grund der Drosselung (, und), die unter beschrieben werden. `ConcurrencyThrottles` `CPUThrottles` `MemoryThrottles` `DiskThrottles` Arten von Metriken für Lambda-Ereignisfunktionen

3. Ausführungsumgebungen pro Funktion

Legen Sie die minimale und maximale Anzahl von Ausführungsumgebungen für Ihre Funktion fest.

Standardverhalten: Das Standardminimum sind 3 Ausführungsumgebungen in Availability Zones, ohne Standardmaximum. Sie können beide Werte überschreiben, nachdem die Funktion erstellt wurde.

Richtlinien für Anpassungen:

- Legen Sie das Mindestmaß fest: Stellen Sie Kapazität für den Basisverkehr bereit und reduzieren Sie die Drosselung bei plötzlichen Ausfällen. Werte unter 3 reduzieren die Redundanz der Availability Zone.
- Legen Sie das Maximum fest: Beschränken Sie die Anzahl der Ausführungsumgebungen, um das Scale-Out zu kontrollieren und Probleme mit störenden Nachbarn zu vermeiden, wenn sich mehrere Funktionen einen Kapazitätsanbieter teilen.
- Funktion deaktivieren: Setzen Sie sowohl das Minimum als auch das Maximum auf 0, um eine Funktion zu deaktivieren, ohne sie zu löschen.

Beispiel:

```
aws lambda put-function-scaling-config \
  --function-name my-lmi-function \
  --qualifier '$LATEST.PUBLISHED' \
  --function-scaling-config MinExecutionEnvironments=5,MaxExecutionEnvironments=20 \
  --region us-east-1
```

Wichtige Hinweise:

- Geltungsbereich der Qualifizierer: Diese Konfigurationen gelten auf Funktionsebene für jeden qualifizierten ARN. Wenn diese Option aktiviert ist`$LATEST.PUBLISHED`, wird die Konfiguration an zukünftige `$LATEST.PUBLISHED` Versionen weitergegeben. Wenn diese Option für eine bestimmte Version festgelegt ist, werden neu veröffentlichte Versionen auf die Standardwerte zurückgesetzt.
- Gepaarte Konfiguration: Sie müssen sowohl den Mindest- als auch den Höchstwert zusammen festlegen. Jede nicht angegebene Einstellung wird auf ihren Standardwert zurückgesetzt. Gültige Werte für beide `MinExecutionEnvironments` und `MaxExecutionEnvironments` liegen im Bereich von 0 bis 15000. Ein Minimum von 0 ist nur gültig, wenn das Maximum ebenfalls 0 ist.
- Auswirkungen auf die Kosten: Die Funktionsdeaktivierung wird auf der Ebene der Funktionsversion wirksam. Lambda beendet eine zugrunde liegende EC2-Instance, sobald sie

keine aktiven Ausführungsumgebungen mehr hat, und die Instance-Gebühren laufen weiter, bis die Kündigung abgeschlossen ist (in der Regel innerhalb weniger Minuten).

Kontrollen auf Ebene des Kapazitätsanbieters

4. Zielressourcenauslastung

Wählen Sie Ihr eigenes Ziel für den CPU-Auslastungsverbrauch.

Standardverhalten: Lambda bietet genügend Spielraum, sodass sich Ihr Traffic innerhalb von 5 Minuten ohne Drosselung verdoppeln kann.

Optimierungsoptionen:

- Wenn Ihre Arbeitslast sehr konstant ist oder wenn Ihre Anwendung nicht auf Drosselungen reagiert, können Sie das Ziel auf ein hohes Niveau setzen, um eine höhere Auslastung und niedrigere Kosten zu erzielen
- Wenn Sie genügend Spielraum für Datenverkehrsschübe haben möchten, können Sie die Ressourcenziele auf ein niedriges Niveau setzen, wodurch mehr Kapazität benötigt wird

5. Auswahl des Instance-Typs

Legen Sie zulässige oder ausgeschlossene Instanztypen fest.

Standardverhalten: Lambda wählt die besten Instanztypen für Ihre Arbeitslast aus. Es wird empfohlen, Lambda Managed Instances die Auswahl der Instanztypen zu überlassen, da die Beschränkung der Anzahl möglicher Instanztypen zu einer geringeren Verfügbarkeit führen kann.

Benutzerdefinierte Konfiguration:

- Spezifische Hardwareanforderungen: Legen Sie für zulässige Instanztypen eine Liste kompatibler Instanzen fest. Wenn Sie beispielsweise eine Anwendung haben, die eine hohe Netzwerkbandbreite benötigt, können Sie mehrere n-Instance-Typen auswählen
- Kostenoptimierung: Für Test- oder Entwicklungsumgebungen können Sie kleinere Instanztypen wie die Instance-Typen m7a.large wählen

Geplante Skalierung

Verwenden Sie Amazon EventBridge Scheduler, um die minimalen und maximalen Ausführungsumgebungen Ihrer Funktion nach einem wiederkehrenden oder einmaligen Zeitplan anzupassen. Dies ist nützlich für vorhersehbare Verkehrsmuster, z. B. beim Hochfahren vor Spitzenzeiten und beim Herunterfahren außerhalb der Spitzenzeiten.

Scheduler-Konfiguration:

- Erstellen Sie eine EventBridge Scheduler-Ausführungsrolle oder verwenden Sie eine vorhandene Rolle, die Ihnen die Erlaubnis erteilt, Ihre Zielfunktion `lambda:PutFunctionScalingConfig` aufzurufen.
- Erstellen Sie einen Zeitplan mithilfe eines Cron- oder Rate-Ausdrucks und zielen Sie dabei auf die `PutFunctionScalingConfig` API als universelles Ziel ab. Geben Sie die neuen `MaxExecutionEnvironments` Werte `MinExecutionEnvironments` und in der Eingabe-Nutzlast an.

Beispiel 1: Skalierung zur Bewältigung des geplanten Spitzenverkehrs

Erstellen Sie zwei Zeitpläne, um vor den Hauptverkehrszeiten nach oben zu skalieren und danach nach unten zu skalieren. Jeder Zeitplan zielt mit aktualisierten `MaxExecutionEnvironments` Werten auf die `PutFunctionScalingConfig` API `MinExecutionEnvironments` ab.

Um 8:00 Uhr UTC hochskalieren (min=100, max=1000):

```
aws scheduler create-schedule \
  --name "ScaleUpLambdaManagedInstances" \
  --schedule-expression "cron(0 8 * * ? *)" \
  --flexible-time-window '{"Mode": "OFF"}' \
  --target '{
    "Arn": "arn:aws:scheduler:::aws-sdk:lambda:PutFunctionScalingConfig",
    "RoleArn": "arn:aws:iam::<account-id>:role/eventbridge-scheduler-role",
    "Input": "{\"FunctionName\": \"my-lmi-function\", \"Qualifier\":
 \"$LATEST.PUBLISHED\", \"FunctionScalingConfig\": {\"MinExecutionEnvironments\": 100,
 \"MaxExecutionEnvironments\": 1000}}"
  }'
```

Um 18:00 Uhr UTC herunterskalieren (min=5, max=20):

```
aws scheduler create-schedule \
```

```
  --name "ScaleDownLambdaManagedInstances" \
  --schedule-expression "cron(0 18 * * ? *)" \
  --flexible-time-window '{"Mode": "OFF"}' \
  --target '{
    "Arn": "arn:aws:scheduler:::aws-sdk:lambda:PutFunctionScalingConfig",
    "RoleArn": "arn:aws:iam::<account-id>:role/eventbridge-scheduler-role",
    "Input": "{\"FunctionName\": \"my-lmi-function\", \"Qualifier\":
 \"$LATEST.PUBLISHED\", \"FunctionScalingConfig\": {\"MinExecutionEnvironments\": 5,
 \"MaxExecutionEnvironments\": 20}}"
  }'
```

Beispiel 2: Außerhalb der Spitzenzeiten deaktivieren und erneut aktivieren

Wenn Sie `MinExecutionEnvironments` sowohl als auch `MaxExecutionEnvironments` auf 0 setzen, wird die Funktionsversion deaktiviert, ohne sie zu löschen. Eine deaktivierte Funktion wird nicht automatisch mit dem Verkehr wieder hochskaliert. Sie müssen sie explizit reaktivieren, indem Sie im Rahmen einer anderen geplanten Aktion Werte festlegen, die nicht Null sind.

Um 22:00 Uhr UTC deaktivieren (min=0, max=0):

```
aws scheduler create-schedule \
  --name "DeactivateLambdaManagedInstances" \
  --schedule-expression "cron(0 22 * * ? *)" \
  --flexible-time-window '{"Mode": "OFF"}' \
  --target '{
    "Arn": "arn:aws:scheduler:::aws-sdk:lambda:PutFunctionScalingConfig",
    "RoleArn": "arn:aws:iam::<account-id>:role/eventbridge-scheduler-role",
    "Input": "{\"FunctionName\": \"my-lmi-function\", \"Qualifier\":
 \"$LATEST.PUBLISHED\", \"FunctionScalingConfig\": {\"MinExecutionEnvironments\": 0,
 \"MaxExecutionEnvironments\": 0}}"
  }'
```

Reaktivieren Sie um 7:00 Uhr UTC (min=10, max=20):

```
aws scheduler create-schedule \
  --name "ReactivateLambdaManagedInstances" \
  --schedule-expression "cron(0 7 * * ? *)" \
  --flexible-time-window '{"Mode": "OFF"}' \
  --target '{
    "Arn": "arn:aws:scheduler:::aws-sdk:lambda:PutFunctionScalingConfig",
    "RoleArn": "arn:aws:iam::<account-id>:role/eventbridge-scheduler-role",
```

```
    "Input": "{\"FunctionName\": \"my-lmi-function\", \"Qualifier\":
\"$LATEST.PUBLISHED\", \"FunctionScalingConfig\": {\"MinExecutionEnvironments\": 10,
\"MaxExecutionEnvironments\": 20}}"
 }'
```

Richtlinien zur Anpassung:

- Erstellen Sie für Workloads mit vorhersehbaren Spitzenwerten mehrere Zeitpläne, die Ihrem Verkehrsmuster entsprechen: einen, um Ihre Funktion vor den Hauptverkehrszeiten zu erhöhen, und einen anderen, um nach den Hauptverkehrszeiten herunterzufahren. Jeder Zeitplan folgt demselben Muster mit aktualisierten `MinExecutionEnvironments` `MaxExecutionEnvironments` AND-Werten.
- Bei der geplanten Skalierung werden die bereitgestellten Unter- und Obergrenzen der Ausführungsumgebungen angepasst, aber die tatsächliche Skalierung zwischen Mindest- und Höchstwert hängt immer noch von der CPU-Auslastung und der gleichzeitigen Auslastung ab.
- Wenn sich Ihr Traffic innerhalb von 5 Minuten nach einer geplanten Skalierung mehr als verdoppelt, kann es bei der Bereitstellung von Kapazität dennoch zu Drosselungen kommen.
- Denken Sie bei der Skalierung auf Null zur Deaktivierung einer Funktion daran, dass für die Reaktivierung ein expliziter Aufruf mit Werten ungleich Null erforderlich ist. `PutFunctionScalingConfig`

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Lesen Sie die laufzeitspezifischen Anleitungen für den Umgang mit Mehrfachparallelität
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Überwachen Sie Skalierungskennzahlen, um das Skalierungsverhalten zu optimieren

Sicherheit und Berechtigungen

Lambda Managed Instances verwenden Kapazitätsanbieter als Vertrauensgrenzen. Lambda-Ereignisfunktionen werden in Containern innerhalb dieser Instanzen ausgeführt, aber Container bieten keine Sicherheitsisolierung zwischen Workloads. Alle Funktionen, die demselben Kapazitätsanbieter zugewiesen sind, müssen gegenseitig vertrauenswürdig sein.

Wichtige Sicherheitskonzepte

- Kapazitätsanbieter: Die Sicherheitsgrenze, die Vertrauensstufen für Lambda-Funktionen definiert
- Container-Isolierung: Container stellen keine Sicherheitsgrenze dar. Verlassen Sie sich nicht auf sie, wenn es um die Sicherheit zwischen nicht vertrauenswürdigen Workloads geht
- Vertrauenstrennung: Trennen Sie Workloads, denen nicht gegenseitig vertraut wird, indem Sie unterschiedliche Kapazitätsanbieter verwenden

Erforderliche Berechtigungen

PassCapacityProvider Aktion

Benutzer benötigen die `lambda:PassCapacityProvider` Erlaubnis, Kapazitätsanbietern Funktionen zuzuweisen. Diese Genehmigung dient als Sicherheitsschleuse und stellt sicher, dass nur autorisierte Benutzer Funktionen bestimmten Kapazitätsanbietern zuweisen können.

Kontoadministratoren kontrollieren über die `lambda:PassCapacityProvider` IAM-Aktion, welche Funktionen bestimmte Kapazitätsanbieter nutzen können. Diese Aktion ist erforderlich, wenn:

- Funktionen erstellen, die Lambda Managed Instances verwenden
- Aktualisierung von Funktionskonfigurationen zur Verwendung eines Kapazitätsanbieters
- Bereitstellung von Funktionen mithilfe von Infrastruktur als Code

Beispiel für eine IAM-Richtlinie

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:PassCapacityProvider",
      "Resource": "arn:aws:lambda:*:*:capacity-provider:trusted-workloads-*"
    }
  ]
}
```

Service-Linked Rolle

AWS Lambda verwendet die `AWSServiceRoleForLambda` serviceverknüpfte Rolle, um die ec2-Ressourcen von Lambda Managed Instances bei Ihren Kapazitätsanbietern zu verwalten.

Bewährte Methoden

1. Nach Vertrauensstufe trennen: Erstellen Sie verschiedene Kapazitätsanbieter für Workloads mit unterschiedlichen Sicherheitsanforderungen
2. Verwenden Sie aussagekräftige Namen: Benennen Sie Kapazitätsanbieter, um deren Verwendungszweck und Vertrauensstufe eindeutig anzugeben (z. B.,`production-trusted`) `dev-sandbox`
3. Least Privilege anwenden: Erteilen Sie `PassCapacityProvider` Berechtigungen nur für die erforderlichen Kapazitätsanbieter
4. Nutzung überwachen: Wird verwendet AWS CloudTrail , um die Zuweisungen und Zugriffsmuster von Kapazitätsanbietern zu überwachen

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Lesen Sie laufzeitspezifische Anleitungen für Java Node.js, und Python

Lambda-Operatorrolle für Lambda Managed Instances

Wenn Sie Lambda Managed Instances verwenden, benötigt Lambda Berechtigungen zur Verwaltung der Rechenkapazität in Ihrem Konto. Die Operatorrolle stellt diese Berechtigungen über IAM-Richtlinien bereit, die es Lambda ermöglichen, EC2-Instances im Kapazitätsanbieter zu verwalten.

Lambda übernimmt bei der Ausführung dieser Verwaltungsvorgänge die Operatorrolle, ähnlich wie Lambda eine Ausführungsrolle übernimmt, wenn Ihre Lambda-Ereignisfunktion ausgeführt wird.

Eine Operatorrolle erstellen

Sie können eine Operatorrolle in der IAM-Konsole oder mit der AWS CLI erstellen. Die Rolle muss Folgendes beinhalten:

- Berechtigungsrichtlinie — Erteilt Berechtigungen zur Verwaltung von Kapazitätsanbietern und zugehörigen Ressourcen
- Vertrauensrichtlinie — Ermöglicht dem Lambda-Dienst (`lambda.amazonaws.com`), die Rolle zu übernehmen

Berechtigungsrichtlinie

Die Operatorrolle benötigt Berechtigungen zur Verwaltung von Kapazitätsanbietern und den zugrunde liegenden Rechenressourcen. Für die Rolle sind mindestens die in der AWSLambdaManagedEC2ResourceOperatorverwalteten Richtlinie festgelegten Berechtigungen erforderlich, derzeit:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateTags",
        "ec2:AttachNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:instance/*",
        "arn:aws:ec2:*:*:network-interface/*",
        "arn:aws:ec2:*:*:volume/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:ManagedResourceOperator": "scaler.lambda.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeAvailabilityZones",
        "ec2:DescribeCapacityReservations",
        "ec2:DescribeInstances",
        "ec2:DescribeInstanceStatus",
        "ec2:DescribeInstanceTypeOfferings",
```

```
        "ec2:DescribeInstanceTypes",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcEncryptionControls"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:image/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:Owner": "amazon"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup"
      ],
      "Resource": "arn:aws:logs:*:*:log-group:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogStream",
        "logs:PutLogEvents"
```

```
      ],
      "Resource": "arn:aws:logs:*:*:log-group:*:log-stream:*"
    }
  ]
}
```

Vertrauensrichtlinie

Die Vertrauensrichtlinie ermöglicht es Lambda, die Betreiberrolle zu übernehmen:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Service-Linked Rolle für verwaltete Lambda-Instanzen

Um den Lebenszyklus von Lambda Managed Instances verantwortungsbewusst zu verwalten, benötigt Lambda dauerhaften Zugriff, um verwaltete Instanzen in Ihrem Konto zu beenden. Lambda verwendet eine dienstgebundene Rolle (SLR) für AWS Identity and Access Management (IAM), um diese Operationen auszuführen.

Automatische Erstellung: Die serviceverknüpfte Rolle wird automatisch erstellt, wenn Sie zum ersten Mal einen Kapazitätsanbieter erstellen. Der Benutzer, der den ersten Kapazitätsanbieter erstellt, muss über die `iam:CreateServiceLinkedRole` Berechtigung für den `lambda.amazonaws.com` Prinzipal verfügen.

Berechtigungen: Die serviceverknüpfte Rolle gewährt Lambda die folgenden Berechtigungen für verwaltete Instanzen:

- `ec2:TerminateInstances`— Um Instanzen am Ende ihres Lebenszyklus zu beenden
- `ec2:DescribeInstances`— Um verwaltete Instanzen aufzuzählen

Löschung: Sie können diese serviceverknüpfte Rolle erst löschen, nachdem Sie alle Kapazitätsanbieter für Lambda Managed Instances in Ihrem Konto gelöscht haben.

Weitere Informationen zu serviceverknüpften Rollen finden Sie unter the section called “Verwenden von servicegebundenen Rollen”.

Grundlegendes zur Lambda Managed Instance-Ausführungsumgebung

Lambda Managed Instances bieten ein alternatives Bereitstellungsmodell, bei dem Ihr Funktionscode auf kundeneigenen Amazon EC2-Instances ausgeführt wird, während Lambda die betrieblichen Aspekte verwaltet. Die Ausführungsumgebung für Managed Instances weist mehrere wichtige Unterschiede zu den Lambda-Funktionen (Standard) auf, insbesondere in der Art und Weise, wie sie gleichzeitige Aufrufe verarbeitet und Container-Lebenszyklen verwaltet.

Hinweis: Informationen zur Lambda-Ausführungsumgebung (Standard) finden Sie unter Grundlegendes zum Lebenszyklus der Lambda-Ausführungsumgebung.

Lebenszyklus der Ausführungsumgebung

Der Lebenszyklus einer Lambda Managed Instances-Funktionsumgebung unterscheidet sich in mehreren wesentlichen Punkten von Lambda (Standard):

Init-Phase

Während der Init-Phase führt Lambda die folgenden Schritte aus:

- Initialisieren und registrieren Sie alle Erweiterungen
- Bootstrappen Sie den Runtime-Einstiegspunkt. Runtime erzeugt die konfigurierte Anzahl von Runtime-Workern (die Implementierung hängt von der Laufzeit ab)
- Führen Sie den Funktionsinitialisierungscode aus (Code außerhalb des Handlers)
- Warten Sie, bis mindestens ein Runtime-Worker die Bereitschaft signalisiert, indem Sie aufrufen `/runtime/invocation/next`

Die Init-Phase gilt als abgeschlossen, wenn die Erweiterungen initialisiert wurden und mindestens ein Runtime-Worker aufgerufen hat. `/runtime/invocation/next` Die Funktion ist dann bereit, Aufrufe zu verarbeiten.

> **Note**
>
> Bei Lambda Managed Instances-Funktionen kann die Initialisierung bis zu 15 Minuten dauern. Das Timeout ist das Maximum von 130 Sekunden oder das konfigurierte Funktions-Timeout (bis zu 900 Sekunden).

Invoke-Phase

Die Aufrufphase für Lambda Managed Instances-Funktionen weist mehrere einzigartige Merkmale auf:

Kontinuierlicher Betrieb. Im Gegensatz zu Lambda (Standard) bleibt die Ausführungsumgebung kontinuierlich aktiv und verarbeitet Aufrufe, sobald sie ankommen, ohne dass zwischen den Aufrufen eingefroren wird.

Parallele Verarbeitung. In derselben Ausführungsumgebung können mehrere Aufrufe gleichzeitig ausgeführt werden, wobei jeder von einem anderen Runtime-Worker verarbeitet wird.

Unabhängige Timeouts. Das konfigurierte Timeout der Funktion gilt für jeden einzelnen Aufruf. Wenn bei einem Aufruf ein Timeout auftritt, markiert Lambda diesen bestimmten Aufruf als fehlgeschlagen, unterbricht jedoch weder andere laufende Aufrufe noch beendet die Ausführungsumgebung.

Umgang mit Gegendruck. Wenn alle Runtime-Worker mit der Verarbeitung von Aufrufen beschäftigt sind, werden neue Aufrufanforderungen abgelehnt, bis ein Worker verfügbar ist.

Fehlerbehandlung und Wiederherstellung

Die Fehlerbehandlung in Lambda Managed Instances-Funktionsumgebungen unterscheidet sich von der in Lambda (Standard):

Runtime-Worker-Fehler. Wenn ein Runtime-Worker-Prozess abstürzt, arbeitet die Ausführungsumgebung mit den verbleibenden gesunden Workern weiter.

Die Erweiterung stürzt ab. Wenn ein Erweiterungsprozess während der Initialisierung oder des Vorgangs abstürzt, wird die gesamte Ausführungsumgebung als fehlerhaft markiert und beendet. Lambda erstellt eine neue Ausführungsumgebung, um sie zu ersetzen.

Nein. reset/repair Im Gegensatz zu Lambda (Standard) versuchen Managed Instances nicht, die Ausführungsumgebung nach Fehlern zurückzusetzen und neu zu initialisieren. Stattdessen werden fehlerhafte Container beendet und durch neue ersetzt.

Timeouts aufrufen

Wenn bei einem einzelnen Aufruf das Zeitlimit überschritten wird, gibt Lambda einen `Task timed out after <timeout> seconds` Fehler mit dem Status Funktionsfehler an den Aufrufer zurück. Lambda Managed Instances beendet Ihren Code jedoch nicht gewaltsam — er wird in der Ausführungsumgebung weiter ausgeführt. Als Funktionsentwickler sind Sie dafür verantwortlich, den Timeout zu erkennen und zu behandeln.

Das Kontextobjekt macht die verbleibende Zeit für den Aufruf verfügbar. Ein Wert von Null oder ein negativer Wert gibt an, dass das Zeitlimit für den Aufruf abgelaufen ist. Andere gleichzeitige Aufrufe in der Ausführungsumgebung setzen die Verarbeitung normal fort.

Wiederholungsverhalten

Wenn ein Aufruf das Zeitlimit überschreitet:

- Synchrone Aufrufe: Der Anrufer erhält den Timeout-Fehler und ist für den erneuten Versuch verantwortlich.
- Asynchrone Aufrufe: Lambda-Wiederholungsversuche basieren auf der Wiederholungsrichtlinie Ihrer Funktion (Standardeinstellung: 2 Wiederholungen). Wenn alle Wiederholungsversuche ausgeschöpft sind, wird das Ereignis an die konfigurierte Warteschlange für unzustellbare Nachrichten oder an das Ziel bei einem Ausfall gesendet, falls vorhanden.
- Zuordnungen der Ereignisquellen: Das Wiederholungsverhalten hängt von der Konfiguration der Ereignisquelle ab (z. B. Stapelgröße, Halbierung bei Fehlern, maximale Anzahl von Wiederholungsversuchen). Der Batch kann je nach Ihren Wiederholungsrichtlinien erneut versucht oder bei einem Ausfall an ein Ziel gesendet werden.

Was passiert, wenn Sie den Timeout nicht einhalten

Wenn Ihr Code die verbleibende Zeit nicht überprüft und die Ausführung stoppt:

- Der Aufruf ist bereits als fehlgeschlagen markiert. Lambda hat bereits einen Timeout-Fehler an den Anrufer zurückgegeben — jede Arbeit, die Ihr Code nach dem Timeout abschließt, geht aus Sicht des Aufrufers effektiv verloren.
- Ressourcen bleiben verbraucht. Ihr Code belegt weiterhin einen Runtime-Worker-Slot, wodurch die Parallelität reduziert wird, die für neue Aufrufe auf dieser Instance verfügbar ist.

- Nichtdeterministisches Verhalten. Ihr Code stoppt nicht, wenn der Timeout ausgelöst wird — er läuft im Hintergrund weiter. Das bedeutet, dass auch dann noch Nebenwirkungen auftreten können, wenn Lambda dem Aufrufer bereits mitgeteilt hat, dass der Aufruf fehlgeschlagen ist.

  Ihr Handler schreibt beispielsweise einen Datensatz in DynamoDB, dann wird der Timeout ausgelöst und Lambda gibt einen Timeout-Fehler an den Aufrufer zurück, aber Ihr Code läuft immer noch und sendet eine SNS-Benachrichtigung. Der Anrufer wiederholt den Aufruf, wodurch der Datensatz erneut geschrieben und die Benachrichtigung erneut gesendet wird. Sie haben jetzt doppelte Daten und doppelte Benachrichtigungen — und es gibt keine einfache Möglichkeit herauszufinden, welche von dem „fehlgeschlagenen“ Aufruf stammen, der noch im Hintergrund lief.

Umgang mit Timeouts in Ihrem Code

Verwenden Sie das Kontextobjekt, um die verbleibende Zeit zu überprüfen und die Verarbeitung vor dem Timeout zu beenden. Konfigurieren Sie einen Puffer, der auf der erwarteten Dauer Ihrer nächsten Arbeitseinheit basiert. Wenn beispielsweise die Verarbeitung jedes Elements etwa 500 ms dauert, legen Sie den Puffer auf mindestens 500 ms plus Spielraum fest.

Sprachspezifische Beispiele für die Behandlung von Timeouts finden Sie im Abschnitt zum Anforderungskontext auf jeder Laufzeitseite:

- Node.js Kontext der Anfrage
- Kontext der Python-Anfrage
- Java-Anforderungskontext
- .NET-Anfragekontext

Lazy-load Container-Images mit Seekable OCI (SOCI)

Bei Lambda Event Functions, die als Container-Images auf Lambda Managed Instances bereitgestellt werden, lädt Lambda das Image auf jeder Instance herunter und entpackt es, bevor eine Ausführungsumgebung gestartet wird. Bei großen Images ist dieser Abruf der größte Teil der Zeit zwischen der Veröffentlichung einer Funktionsversion oder der Skalierung und der Verarbeitung des ersten Aufrufs durch die Ausführungsumgebung. Mit Seekable OCI (SOCI) kann Lambda die Ausführungsumgebung starten, bevor das gesamte Image heruntergeladen wurde.

SOCI ist ein Open-Source-Technologie entwickelt von AWS. Es verwendet einen Index, der jede Datei im Bild ihrer Position in den Bildebenen zuordnet. Lambda verwendet den Index, um bei Bedarf nur die Dateien abzurufen, während Ihre Funktion sie liest, anstatt das gesamte Bild herunterzuladen

und zu entpacken, bevor der Container gestartet wird. Container-Images bündeln in der Regel eine Vielzahl von Bibliotheken und Abhängigkeiten, von denen nicht alle beim Start oder bei einem einzelnen Aufruf gelesen werden. Lazy Loading macht sich dies zunutze, indem es nur das abruft, was Ihre Funktion tatsächlich liest, wodurch die Zeit bis zum ersten Aufruf reduziert wird und die Startzeit viel weniger empfindlich auf die Gesamtgröße des Bilds reagiert. Der Vorteil ist am größten bei Bildern mit mehreren Gigabyte, bei denen Ihre Funktion pro Aufruf nur eine Teilmenge des Inhalts liest, z. B. bei Bildern, die Frameworks für maschinelles Lernen enthalten.

So funktioniert SOCI auf Lambda Managed Instances

1. Sie erstellen einen SOCI-Index für Ihr Image und übertragen ihn zusammen mit dem Image an Amazon ECR. Das Ergebnis ist ein OCI-Image-Index (Open Container Initiative), der auf zwei Artefakte verweist: Ihr ursprüngliches Image-Manifest und den SOCI-Index.
2. Sie erstellen oder aktualisieren Ihre Funktion mit einer Bild-URI, die auf den Bildindex verweist. Lambda erkennt den SOCI-Index und aktiviert verzögertes Laden für die Ausführungsumgebungen dieser Funktion. Sie müssen keine andere Funktionskonfiguration ändern.
3. Lambda ruft Bildinhalte bei Bedarf ab. Dateien, die Ihr Initialisierungscode liest, werden während der Initialisierung abgerufen. Dateien, die Ihr Handler zum ersten Mal liest, werden während dieses Aufrufs abgerufen. Lambda lädt den Rest des Bilds im Hintergrund nicht herunter.

> **Note**
>
> Wenn Ihre Funktion eine Datei zum ersten Mal während eines Aufrufs und nicht während der Initialisierung liest, ruft Lambda diese Datei beim ersten Aufruf ab, wodurch die Dauer dieses Aufrufs verlängert wird. Um die Aufrufdauer konsistent zu halten, importieren Sie alle Bibliotheken und lesen Sie alle Dateien, die Ihre Funktion benötigt, in Ihrem Initialisierungscode, nicht im Handler. Wir empfehlen, Ihre Funktion mit und ohne SOCI zu testen, um die Startzeit und die Aufrufdauer für Ihren Workload zu vergleichen.

Erstellen Sie einen SOCI-Index für Ihr Bild

Das folgende Beispiel läuft auf Amazon Linux 2023. Es installiert den SOCI-Snapshotter, containerd und nerdctl, erstellt dann einen SOCI-Index für ein vorhandenes Bild in Amazon ECR und überträgt das Ergebnis. Ersetzen Sie die Platzhalter-Werte durch Ihre eigenen. Weitere Möglichkeiten, einen SOCI-Index zu erstellen und zu übertragen, finden Sie unter Erste Schritte mit dem SOCI-Snapshotter auf GitHub.

```
sudo yum install -y soci-snapshotter containerd nerdctl
sudo systemctl start soci-snapshotter
sudo systemctl restart containerd

ACCOUNT_ID="111122223333"
REGION="us-east-2"
REPOSITORY_NAME="my-function"
ORIGINAL_IMAGE_TAG="v1"
SOCI_IMAGE_TAG="v1-soci"
REGISTRY="$ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com"

aws ecr get-login-password --region $REGION | sudo nerdctl login -u AWS --password-
stdin $REGISTRY

sudo nerdctl pull $REGISTRY/$REPOSITORY_NAME:$ORIGINAL_IMAGE_TAG
sudo nerdctl image convert --soci $REGISTRY/$REPOSITORY_NAME:$ORIGINAL_IMAGE_TAG
 $REGISTRY/$REPOSITORY_NAME:$SOCI_IMAGE_TAG
sudo nerdctl push --platform linux/arm64 $REGISTRY/$REPOSITORY_NAME:$SOCI_IMAGE_TAG
```

> **Note**
>
> Dieses Beispiel erstellt eine `arm64` (AWS Graviton) -Bild. Lambda unterstützt nur Container-Images mit einer einzigen Architektur, daher muss Ihr Image auf eine Architektur mit einem einzigen Befehlssatz abzielen, und `--platform` Der Wert muss mit der Architektur Ihrer Funktion übereinstimmen. Weitere Informationen finden Sie unter the section called “Befehlssätze (ARM/x86)”.

Benutzen `$REGISTRY/$REPOSITORY_NAME:$SOCI_IMAGE_TAG` als Bild-URI, wenn Sie Ihre Funktion erstellen oder aktualisieren. Veröffentlichen Sie dann eine Funktionsversion. Lambda löst die Funktionsversion in den Bildindex auf und ermöglicht verzögertes Laden.

Anforderungen und Einschränkungen

Version des SOCI-Indexes

Lambda unterstützt nur SOCI Index Manifest v2 (Medientyp) `application/vnd.amazon.soci.index.v2+json`). Lambda unterstützt keine SOCI Index Manifest v1-Indizes. Das `nerdctl image convert --soci` Der Befehl im vorherigen Abschnitt erzeugt einen v2-Index.

Format des Bildindex

Der Bildindex muss genau zwei Einträge enthalten: den SOCI-Index und ein OCI-Image-Manifest.

Image-URI

Zeigen Sie mit der Bild-URI Ihrer Funktion auf den SOCI-indexed Tag (zum Beispiel `my-function:v1-soci`), nicht am ursprünglichen Bild-Tag. Lambda löst die Funktionsversion in den Bildindex auf. Um das Bild oder den Index zu ändern, drücken Sie einen neuen SOCI-indexed taggen und aktualisieren Sie die Bild-URI der Funktion darauf.

Verfügbarkeit

SOCI ist nur für Container-Image-Funktionen verfügbar, die auf Lambda Managed Instances ausgeführt werden.

LATEST.PUBLISHED $-Version in von Lambda verwalteten Instances

Die Funktionen von Lambda Managed Instances unterstützen den gleichen Workflow zur nummerierten Versionierung wie Lambda (Standard). Wenn Sie es vorziehen, keine nummerierten Versionen zu verwalten, führt Lambda Managed Instances einen neuen Versionstyp ein:. `$LATEST.PUBLISHED` Mit diesem Versionstyp können Sie eine veröffentlichte Version mit aktualisiertem Code oder aktualisierter Konfiguration erstellen oder erneut veröffentlichen. Sie müssen keine nummerierten Versionen verwalten.

Wichtiger Unterschied zu $LATEST: Wenn Sie eine Lambda Managed Instances-Funktion mit einem unqualifizierten ARN aufrufen, ruft Lambda implizit die Version und nicht die unveröffentlichte `$LATEST.PUBLISHED` $LATEST-Version auf.

Mit dem folgenden CLI-Befehl wird die Version erstellt oder erneut veröffentlicht. AWS `$LATEST.PUBLISHED`

```
aws lambda publish-version --function-name my-function --publish-to LATEST_PUBLISHED
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST.PUBLISHED",
  "Version": "$LATEST.PUBLISHED",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
```

```
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> Wenn Sie AWS CloudFormation oder die Lambda-Konsole verwenden, um eine Lambda Managed Instances-Funktion zu erstellen, erstellt Lambda die Version automatisch. `$LATEST.PUBLISHED`

Funktionen von Lambda Managed Instances bereitstellen und aktualisieren

Wenn Sie eine Lambda Managed Instances-Funktion aktualisieren, stellt Lambda Ausführungsumgebungen für die neue Funktionsversion auf Ihren verwalteten Instances bereit, bevor der Datenverkehr dorthin verlagert wird. Sie können die Geschwindigkeit und Sicherheit der Bereitstellung kontrollieren, indem Sie eine Bereitstellungsstrategie auswählen. In diesem Thema werden die Bereitstellungsstrategien beschrieben, die für Lambda Managed Instances-Funktionen verfügbar sind, und wie sie mithilfe von Infrastructure as Code (IaC) definiert werden.

> **Note**
>
> Bevor Sie eine Funktion bereitstellen, benötigen Sie einen Kapazitätsanbieter. Weitere Informationen finden Sie unter Kapazitätsanbieter für Lambda Managed Instances.

Bereitstellungsstrategien

Lambda Managed Instances unterstützt die folgenden Strategien für die Veröffentlichung einer neuen Funktionsversion. Wählen Sie die Strategie, die dem Umfang der Kontrolle entspricht, die Sie über den Rollout behalten möchten. Fortlaufende Updates sind die einfachste Option. Blue/green und Canary-Deployments geben dir mehr Kontrolle darüber, wie der Traffic auf die neue Version übertragen wird.

Fortlaufende Updates

Um ein fortlaufendes Update durchzuführen, veröffentlichen Sie Ihre Änderungen im `$LATEST.PUBLISHED` Version Ihrer Funktion. `$LATEST.PUBLISHED` ist ein für Lambda Managed

Instances spezifischer Versionstyp, den Sie mit neuem Code oder neuer Konfiguration erneut veröffentlichen, anstatt nummerierte Versionen beizubehalten. Wenn Sie eine Lambda Managed Instances-Funktion mit ihrem unqualifizierten Amazon-Ressourcennamen (ARN) aufrufen, ruft Lambda die `$LATEST.PUBLISHED` Version. Weitere Informationen finden Sie unter the section called “Veröffentlichung der Version”.

Jedes Mal, wenn Sie erneut veröffentlichen `$LATEST.PUBLISHED`, Lambda ersetzt die Ausführungsumgebungen auf Ihren verwalteten Instances schrittweise durch die neue Version, ohne dass Sie den Traffic direkt über einen Funktionsalias verwalten müssen. Dies ist die einfachste Strategie.

Verwenden Sie fortlaufende Updates, wenn Sie eine automatische schrittweise Verlagerung des Datenverkehrs von der alten Version auf die neue Version wünschen und keine direkte Kontrolle über den Prozentsatz des Datenverkehrs benötigen, der an jede Version gesendet wird, um die Bereitstellung oder das Rollback zu verwalten.

Blue/green Bereitstellungen mit einem gewichteten Alias

In einem blue/green Bei einer Bereitstellung veröffentlichen Sie die neue Version zusammen mit der aktuellen Version und verwenden eine Alias mit einer Routing-Konfiguration, um einen Prozentsatz des Datenverkehrs an die neue Version zu senden. Sie rufen die Funktion über den Alias-qualifizierten ARN auf. Um die Bereitstellung weiterzuentwickeln, erhöhen Sie die Gewichtung der neuen Version. Um sie rückgängig zu machen, verlagern Sie die Gewichtung auf die vorherige Version zurück.

Das folgende Beispiel verwendet AWS CLI, um 10 Prozent des Datenverkehrs an Version zu senden 2 während der Version 1 bedient weiterhin die restlichen 90 Prozent.

```
aws lambda update-alias \
  --function-name my-managed-instance-function \
  --name production \
  --function-version 1 \
  --routing-config AdditionalVersionWeights={"2"=0.1}
```

In diesem Befehl gilt Folgendes:

- `my-managed-instance-function` ist der Name Ihrer Funktion.
- `production` ist der Name des Alias, den Sie aufrufen.
- `--function-version 1` ist die Version, die weiterhin den verbleibenden Datenverkehr empfängt.

- `AdditionalVersionWeights={"2"=0.1}` sendet 10 Prozent des Datenverkehrs an Version 2.

Um ein Rollback durchzuführen, löschen Sie die Routing-Konfiguration, sodass der gesamte Datenverkehr zur stabilen Version zurückkehrt.

```
aws lambda update-alias \
  --function-name my-managed-instance-function \
  --name production \
  --function-version 1 \
  --routing-config '{}'
```

In diesem Befehl `--routing-config '{}'` löscht die Routing-Konfiguration, sodass der gesamte Datenverkehr zur Version in zurückkehrt `--function-version`, Version `1`.

Canary-Einsätze mit AWS CodeDeploy

Um eine zu automatisieren blue/green Verkehrsverlagerung, benutze AWS CodeDeploy. CodeDeploy leitet den Datenverkehr gemäß einer Bereitstellungskonfiguration von der aktuellen Version auf die neue Version um. Beispielsweise werden 10 Prozent des Datenverkehrs an die neue Version gesendet, fünf Minuten gewartet und dann der Rest verschoben. Wenn ein CloudWatch ein mit dem Einsatz verbundener Alarm während der Schicht in einen Alarmzustand übergeht, CodeDeploy setzt den Einsatz automatisch zurück.

Für weitere Informationen über CodeDeploy Bereitstellungskonfigurationen für Lambda finden Sie unter Bereitstellungskonfigurationen auf einer Lambda-Rechenplattform in der AWS CodeDeploy Benutzerleitfaden.

Wählen Sie zwischen $LATEST.PUBLISHED und numerische Versionen mit Aliasen

Bei beiden Bereitstellungsmodellen werden Änderungen eingeführt allmählich. Ob du aktualisierst `$LATEST.PUBLISHED` Wenn Sie den Traffic mit einem gewichteten Alias verlagern, ersetzt Lambda Managed Instances die Ausführungsumgebungen der alten Version schrittweise durch die neue Version und nicht alle gleichzeitig. Dieser Ansatz sorgt für eine effiziente Nutzung der Instanzen während des Rollouts. Anstatt im Voraus einen kompletten zweiten Satz von Ausführungsumgebungen bereitzustellen, ersetzt Lambda diese schrittweise: Bei jedem Schritt gewinnt es Kapazität aus der alten Version zurück, bevor der nächste Batch der neuen Version veröffentlicht wird, sodass beim Rollout vermieden wird, dass die doppelte Basiskapazität erreicht wird.

Wie $LATEST.PUBLISHED Der Rollout schreitet voran

Wenn Sie ein Update veröffentlichen für `$LATEST.PUBLISHED`, Lambda beginnt mit der schrittweisen Einführung der neuen Version. Bei jedem Schritt erstellt Lambda neue Ausführungsumgebungen — bis zu 20 Prozent der Gesamtmenge der alten Version — und ersetzt schrittweise die Ausführungsumgebungen der alten Version.

Zwei Faktoren bestimmen, wie lange die Einführung dauert:

- Zeit der Initialisierung - Jeder Schritt wartet darauf, dass die neuen Ausführungsumgebungen initialisiert sind, bevor er fortgesetzt wird. Wenn die Initialisierung 10 Minuten dauert, wartet jeder Schritt mindestens 10 Minuten.
- Dauer des Aufrufs — Bevor Lambda eine alte Ausführungsumgebung entfernen kann, müssen die während des Fluges ausgeführten Aufrufe abgeschlossen sein (Drain). Wenn Ihre durchschnittliche Aufrufdauer 5 Minuten beträgt, dauert das Entleeren bei jedem Schritt im Durchschnitt etwa 5 Minuten.

Da Lambda Managed Instances für eine hohe Instanzauslastung sorgen, tragen sowohl die Initialisierung als auch die Entleerung zur Gesamtbereitstellungszeit bei. Funktionen mit langer Initialisierung oder lang andauernden Aufrufen benötigen länger, bis sie vollständig eingeführt werden.

Wie entwickeln sich numerische Versionen mit Aliasen

Gewichtete Alias-Aktualisierungen erfolgen auf die gleiche Weise schrittweise. Wenn Sie die Routing-Gewichtungen für einen Alias ändern — beispielsweise von 90 Prozent Version 1 und 10 Prozent Version 2 auf 100 Prozent — ersetzt Lambda Managed Instances langsam die Ausführungsumgebungen der Version 1 durch Version 2, ohne dass überschüssige Instance-Kapazität bereitgestellt wird.

Verwenden Sie numerische Versionen mit einem Alias, wenn Sie eine genauere Kontrolle über den schrittweisen Rollout und die versionsspezifische Beobachtbarkeit wünschen. Da es sich bei jeder Version um einen eigenen Qualifizierer handelt, können Sie die Kennzahlen für die Canary-Version und die Stable-Version vergleichen, bevor Sie für eine Änderung werben. Weitere Informationen finden Sie unter Metriken für Ihre Lambda-Funktionen anzeigen.

Konfiguration der Skalierung nach einer Bereitstellung

Jede Funktionsversion, die Sie bereitstellen, hat ihre eigene Skalierungskonfiguration, die die minimale und maximale Anzahl von Ausführungsumgebungen steuert, die Lambda für diese Version verwaltet. Stellen Sie die Skalierungskonfiguration ein, nachdem Sie eine Version veröffentlicht haben, sodass die neue Version den Datenverkehr mit der erwarteten Kapazität bedienen kann.

Im folgenden Beispiel wird die Skalierungskonfiguration für eine bestimmte Version festgelegt. Ersetzen 2 mit deiner Versionsnummer oder mit `$LATEST.PUBLISHED`.

```
aws lambda put-function-scaling-config \
  --function-name my-managed-instance-function \
  --qualifier 2 \
  --function-scaling-config MinExecutionEnvironments=3,MaxExecutionEnvironments=20
```

Verwaltung von Bereitstellungen mit Infrastruktur als Code

Sie können die Funktionen von Lambda Managed Instances und deren Kapazitätsanbieter mit Infrastruktur als Code definieren und bereitstellen, einschließlich AWS CloudFormation, der AWS Cloud-Entwicklungskit (AWS CDK) und Tools von Drittanbietern wie HashiCorp Terraform. Modellieren Sie Ihren Kapazitätsanbieter, Ihre Funktion, Ihre Versionen und Ihren Alias als Ressourcen, sodass Ihre Bereitstellungsstrategie reproduzierbar und überprüfbar ist.

Beachten Sie bei der Verwaltung von Lambda Managed Instance-Bereitstellungen mit IaC die folgenden Überlegungen:

- Veröffentlichen Sie eine Version, um eine Änderung zu aktivieren. Eine neue Funktionsversion wird auf den Instanzen des Kapazitätsanbieters erst aktiv, nachdem sie veröffentlicht wurde. Modellieren Sie jede Version, die Sie weiterhin verwenden möchten, z. B. beide Seiten einer blue/green shift — als eigene Ressource.
- Die Skalierung der Konfiguration erfolgt pro Version. Wenden Sie die minimalen und maximalen Einstellungen für die Ausführungsumgebung auf jede Version an oder `$LATEST.PUBLISHED`. Wenn Ihr IAC-Tool die Skalierungskonfiguration nicht direkt modelliert, stellen Sie sie mit dem `put-function-scaling-config` AWS CLI-Befehl nach der Bereitstellung.
- Bereitstellungspakete müssen von Amazon S3 stammen, wenn Sie das verwenden AWS Cloud Control-API. Die AWS Die Cloud Control API akzeptiert den Code einer Funktion nur als Amazon S3-Standort- oder Container-Image-URI, nicht als lokal hochgeladene ZIP-Datei. Wenn Ihr IAC-Tool diese API verwendet, laden Sie Ihr Bereitstellungspaket zuerst in einen S3-Bucket hoch und

referenzieren Sie es mit Bucket und Schlüssel. Verwenden Sie einen versionierten Bucket, damit Aktualisierungen des Objekts erkannt werden.

Beispiel: CloudFormation Muster

Das Folgende CloudFormation Die Vorlage zeigt eine blue/green Einsatz. Es erstellt einen Kapazitätsanbieter, eine Lambda Managed Instances-Funktion, zwei veröffentlichte Versionen (jede mit ihrer eigenen Funktionsskalierungskonfiguration) und einen Alias mit dem Namen `production` das leitet 90 Prozent des Datenverkehrs an die stabile Version und 10 Prozent an die Canary-Version weiter. Um für den Canary zu werben, legen Sie den Alias fest `FunctionVersion` zur Canary-Version und entferne den `RoutingConfig`. Um den Vorgang rückgängig zu machen, entfernen Sie den `RoutingConfig` damit der gesamte Verkehr zur stabilen Version zurückkehrt. Ersetzen Sie das Subnetz, die Sicherheitsgruppe, die Rolle und die Amazon S3-Werte durch Ihre eigenen Werte.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda Managed Instances blue/green deployment with a weighted alias

Resources:
  MyCapacityProvider:
    Type: AWS::Lambda::CapacityProvider
    Properties:
      CapacityProviderName: my-capacity-provider
      VpcConfig:
        SubnetIds:
          - subnet-0123456789abcdef0
        SecurityGroupIds:
          - sg-0123456789abcdef0
      PermissionsConfig:
        CapacityProviderOperatorRoleArn: arn:aws:iam::111122223333:role/my-capacity-
provider-operator-role
      InstanceRequirements:
        Architectures:
          - arm64

  MyFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-managed-instance-function
      Role: arn:aws:iam::111122223333:role/my-function-execution-role
      Runtime: python3.13
      Handler: index.lambda_handler
```

```
      Architectures:
        - arm64
      Code:
        S3Bucket: my-deployment-bucket
        S3Key: function.zip
      CapacityProviderConfig:
        LambdaManagedInstancesCapacityProviderConfig:
          CapacityProviderArn: !GetAtt MyCapacityProvider.Arn

  # Stable version - receives the majority of traffic
  StableVersion:
    Type: AWS::Lambda::Version
    Properties:
      FunctionName: !Ref MyFunction
      FunctionScalingConfig:
        MinExecutionEnvironments: 10
        MaxExecutionEnvironments: 50

  # Canary version - receives a small percentage of traffic
  CanaryVersion:
    Type: AWS::Lambda::Version
    Properties:
      FunctionName: !Ref MyFunction
      FunctionScalingConfig:
        MinExecutionEnvironments: 3
        MaxExecutionEnvironments: 20

  ProductionAlias:
    Type: AWS::Lambda::Alias
    Properties:
      FunctionName: !Ref MyFunction
      Name: production
      FunctionVersion: !GetAtt StableVersion.Version
      RoutingConfig:
        AdditionalVersionWeights:
          - FunctionVersion: !GetAtt CanaryVersion.Version
            FunctionWeight: 0.1
```

Beispiel: CDK

Das folgende CDK-Beispiel (TypeScript) definiert denselben Kapazitätsanbieter und dieselbe Funktion unter Verwendung der Level-1-Konstrukte (L1) aus `aws-cdk-lib/aws-lambda` Modul. `operatorRole` und `executionRole` sind IAM-Rollen an anderer Stelle in Ihrem Stack definiert.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';

const capacityProvider = new lambda.CfnCapacityProvider(this, 'MyCapacityProvider', {
  capacityProviderName: 'my-capacity-provider',
  vpcConfig: {
    subnetIds: ['subnet-0123456789abcdef0'],
    securityGroupIds: ['sg-0123456789abcdef0'],
  },
  permissionsConfig: {
    capacityProviderOperatorRoleArn: operatorRole.roleArn,
  },
  instanceRequirements: {
    architectures: ['arm64'],
  },
});

const fn = new lambda.CfnFunction(this, 'MyFunction', {
  functionName: 'my-managed-instance-function',
  role: executionRole.roleArn,
  runtime: 'python3.13',
  handler: 'index.lambda_handler',
  architectures: ['arm64'],
  code: { s3Bucket: 'my-deployment-bucket', s3Key: 'function.zip' },
  capacityProviderConfig: {
    lambdaManagedInstancesCapacityProviderConfig: {
      capacityProviderArn: capacityProvider.attrArn,
    },
  },
  functionScalingConfig: {
    minExecutionEnvironments: 3,
    maxExecutionEnvironments: 20,
  },
  publishToLatestPublished: true,
});
fn.addDependency(capacityProvider);
```

Beispiel: Canary-Bereitstellung mit AWS CodeDeploy

Das folgende CDK-Beispiel (TypeScript) verwendet AWS CodeDeploy um den Verkehr automatisch auf eine neue Funktionsversion umzuleiten. Das `LambdaDeploymentGroup` sendet fünf Minuten lang 10 Prozent des Datenverkehrs auf die neue Version und verschiebt dann den Rest. Wenn

der CloudWatch während des Einsatzes in einen Alarmzustand übergeht, CodeDeploy kehrt zur vorherigen Version zurück.

`fn` ist der L1 `CfnFunction` aus dem vorherigen Beispiel. Weil CodeDeployist `LambdaDeploymentGroup` arbeitet mit dem Level-2-Alias (L2) und den Funktionskonstrukten, importiert die L1-Funktion per ARN und erstellt daraus die Version und den Alias.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as codedeploy from 'aws-cdk-lib/aws-codedeploy';
import * as cloudwatch from 'aws-cdk-lib/aws-cloudwatch';

// Import the L1 function so that L2 constructs can reference it.
const importedFn = lambda.Function.fromFunctionArn(this, 'ImportedFunction',
 fn.attrArn);

// Publish the version that CodeDeploy shifts traffic to.
const canaryVersion = new lambda.CfnVersion(this, 'CanaryVersion', {
  functionName: fn.ref,
});

// Create the alias that CodeDeploy shifts traffic on.
const alias = new lambda.Alias(this, 'ProductionAlias', {
  aliasName: 'production',
  version: lambda.Version.fromVersionAttributes(this, 'CanaryVersionRef', {
    lambda: importedFn,
    version: canaryVersion.attrVersion,
  }),
});

// Roll back automatically if this alarm fires during the deployment.
const errorAlarm = new cloudwatch.Alarm(this, 'ErrorAlarm', {
  metric: importedFn.metricErrors({ period: cdk.Duration.minutes(1) }),
  threshold: 5,
  evaluationPeriods: 1,
  comparisonOperator: cloudwatch.ComparisonOperator.GREATER_THAN_OR_EQUAL_TO_THRESHOLD,
});

new codedeploy.LambdaDeploymentGroup(this, 'CanaryDeployment', {
  alias,
  deploymentConfig: codedeploy.LambdaDeploymentConfig.CANARY_10PERCENT_5MINUTES,
  alarms: [errorAlarm],
  autoRollback: {
```

```
    failedDeployment: true,
    deploymentInAlarm: true,
  },
});
```

Nächste Schritte

- Verstehen Skalierung für von Lambda verwaltete Instances
- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Überprüfung bewährte Methoden für von Lambda verwaltete Instances

Laufzeiten von Lambda Managed Instances

Lambda-Ereignisfunktionen auf Lambda Managed Instances unterstützen beides `.zip` Pakete für die Bereitstellung von Dateiarchiven und Container-Images, die das Format der Open Container Initiative (OCI) verwenden. Informationen zur Verkürzung der Startzeit für große Container-Images finden Sie unter Lazy-load Container-Images mit Seekable OCI (SOCI).

Lambda verarbeitet Anfragen unterschiedlich, wenn Lambda Managed Instances verwendet werden. Anstatt Anfragen nacheinander in jeder Ausführungsumgebung zu bearbeiten, verarbeiten Lambda Managed Instances mehrere Anfragen gleichzeitig in jeder Ausführungsumgebung. Diese Änderung des Ausführungsmodells bedeutet, dass Lambda Event Functions, die Lambda Managed Instances verwenden, Thread-Sicherheit, Statusmanagement und Kontextisolierung berücksichtigen müssen. Diese Bedenken treten im Lambda-Single-Concurrency-Modell (Standard) nicht auf. Darüber hinaus variiert die Implementierung mit mehreren Parallelitäten zwischen den Laufzeiten.

Unterstützte Sprachen

Lambda Managed Instances können mit den folgenden Programmiersprachen und Laufzeiten verwendet werden:

- Java: Java 21 und später.
- Python: Python 3.13 und später.
- Node.js: Node.js 22 und später.
- .NET: .NET 8 und später.
- Rost: Unterstützt mit der OS-only Laufzeit `provided.al2023` und später.

Language-specific Überlegungen

Jede Programmiersprache implementiert die Mehrfachparallelität anders. Sie müssen verstehen, wie Mehrfachparallelität in der von Ihnen gewählten Programmiersprache implementiert wird, um die entsprechenden Best Practices für Parallelität anwenden zu können.

Java

Verwendet einen einzigen Prozess mit Betriebssystem-Threads für die Parallelität. Mehrere Threads führen die Handler-Methode gleichzeitig aus, was eine threadsichere Behandlung von Zustands- und gemeinsam genutzten Ressourcen erfordert.

Python

Verwendet mehrere Python-Prozesse, bei denen jede gleichzeitige Anforderung in einem separaten Prozess ausgeführt wird. Dies schützt vor den meisten Parallelitätsproblemen, obwohl bei gemeinsam genutzten Ressourcen wie dem `/tmp` Verzeichnis.

Node.js

Nutzt Worker-Threads mit asynchroner Ausführung. Gleichzeitige Anforderungen werden auf Worker-Threads verteilt, und jeder Worker-Thread kann auch gleichzeitige Anforderungen asynchron verarbeiten, was eine sichere Handhabung von Zustands- und gemeinsam genutzten Ressourcen erfordert.

.NET

Verwendet .NET-Aufgaben mit der asynchronen Verarbeitung mehrerer gleichzeitiger Anfragen. Erfordert den sicheren Umgang mit Zustands- und gemeinsam genutzten Ressourcen.

Rust

Verwendet einen einzelnen Prozess mit asynchronen Aufgaben, unterstützt von Tokio. Der Handler muss `Clone` + `Send`.

Nächste Schritte

Detaillierte Informationen zu den einzelnen Laufzeiten finden Sie in den folgenden Themen:

- Java-Laufzeit für von Lambda verwaltete Instances
- Node.js Laufzeit für von Lambda verwaltete Instances
- Python-Laufzeit für von Lambda verwaltete Instances

- .NET-Laufzeit für von Lambda verwaltete Instanzen
- Rust-Unterstützung für von Lambda verwaltete Instances

Java-Laufzeit für Lambda Managed Instances

Für Java-Laufzeiten verwenden Lambda Managed Instances Betriebssystem-Threads aus Gründen der Parallelität. Lambda lädt Ihr Handler-Objekt während der Initialisierung einmal pro Ausführungsumgebung und erstellt dann mehrere Threads. Diese Threads werden parallel ausgeführt und erfordern eine threadsichere Handhabung von Zustands- und gemeinsam genutzten Ressourcen. Jeder Thread verwendet dasselbe Handler-Objekt und alle statischen Felder.

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anforderungen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für Java-Laufzeiten ist der Standardwert 32 gleichzeitige Anforderungen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Dieser Wert bestimmt auch die Anzahl der Threads, die von der Java-Laufzeit verwendet werden. Lambda passt die Anzahl der gleichzeitigen Anforderungen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität der einzelnen Ausführungsumgebungen, diese Anforderungen zu verarbeiten.

Erstellung von Funktionen für Mehrfachparallelität

Sie sollten bei der Verwendung von Lambda Managed Instances dieselben Thread-Sicherheitspraktiken anwenden wie in jeder anderen Multithread-Umgebung. Da das Handler-Objekt von allen Runtime-Worker-Threads gemeinsam genutzt wird, muss jeder veränderbare Status threadsicher sein. Dazu gehören Sammlungen, Datenbankverbindungen und alle statischen Objekte, die während der Anforderungsverarbeitung geändert werden.

AWS SDK-Clients sind threadsicher und erfordern keine besondere Behandlung.

Beispiel: Datenbank-Verbindungspools

Der folgende Code verwendet ein statisches Datenbankverbindungsobjekt, das von Threads gemeinsam genutzt wird. Abhängig von der verwendeten Verbindungsbibliothek ist dies möglicherweise nicht threadsicher.

```
public class DBQueryHandler implements RequestHandler<Object, String> {
```

```
    // Single connection shared across all threads - NOT SAFE
    private static Connection connection;

    public DBQueryHandler() {
        this.connection = DriverManager.getConnection(jdbcUrl, username, password);
    }

    @Override
    public String handleRequest(Object input, Context context) {
        PreparedStatement stmt = connection.prepareStatement(query);
        ResultSet rs = stmt.executeQuery();
        // Multiple threads using same connection causes issues
        return result.toString();
    }
}
```

Ein threadsicherer Ansatz besteht darin, einen Verbindungspool zu verwenden. Im folgenden Beispiel ruft der Funktionshandler eine Verbindung aus dem Pool ab. Die Verbindung wird nur im Kontext einer einzelnen Anfrage verwendet.

```
public class DBQueryHandler implements RequestHandler<Object, String> {

    private static HikariDataSource dataSource;

    public DBQueryHandler() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/your_database");
        dataSource = new HikariDataSource(config); // Create pool once per Lambda
 container
    }

    @Override
    public String handleRequest(Object input, Context context) {
        String query = "SELECT column_name FROM your_table LIMIT 10";
        StringBuilder result = new StringBuilder("Data:\n");

        // try-with-resources automatically calls close() on the connection,
        // which returns it to the HikariCP pool (does NOT close the physical DB
 connection)
        try (Connection connection = dataSource.getConnection();
             PreparedStatement stmt = connection.prepareStatement(query);
             ResultSet rs = stmt.executeQuery()) {
```

```
            while (rs.next()) {
                result.append(rs.getString("column_name")).append("\n");
            }

        } catch (Exception e) {
            context.getLogger().log("Error: " + e.getMessage());
            return "Error";
        }

        return result.toString();
    }
}
```

Beispiel: Sammlungen

Standard-Java-Sammlungen sind nicht threadsicher:

```
public class Handler implements RequestHandler<Object, String> {
    private static List<String> items = new ArrayList<>();
    private static Map<String, Object> cache = new HashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Not thread-safe
        cache.put("key", input); // Not thread-safe
        return "Success";
    }
}
```

Verwenden Sie stattdessen threadsichere Sammlungen:

```
public class Handler implements RequestHandler<Object, String> {
    private static final List<String> items =
        Collections.synchronizedList(new ArrayList<>());
    private static final ConcurrentHashMap<String, Object> cache =
        new ConcurrentHashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Thread-safe
        cache.put("key", input); // Thread-safe
        return "Success";
    }
```

```
}
```

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anforderungen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn ein anderer Prozess die Datei überschreibt. Um dieses Problem zu beheben, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Thread oder pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz ausgeschöpft wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge aus verschiedenen Anfragen werden in Protokollen verschachtelt) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal.

Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. ??? Dieses Format beinhaltet das`requestId`, sodass Logeinträge einer einzelnen Anfrage zugeordnet werden können. Wenn Sie das `LambdaLogger` Objekt aus `context.getLogger()` dem verwenden, `requestId` ist es automatisch in jedem Protokolleintrag enthalten. Weitere Informationen finden Sie unterthe section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java”.

Kontext anfordern

Das `context` Objekt ist an den Anforderungsthread gebunden. Using `context.getAwsRequestId()` bietet threadsicheren Zugriff auf die Anforderungs-ID für die aktuelle Anfrage.

Wird für `context.getXrayTraceId()` den Zugriff auf die X-Ray Trace-ID verwendet. Dies bietet Thread-sicheren Zugriff auf die Trace-ID für die aktuelle Anfrage. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable mit Lambda Managed Instances nicht. Die X-Ray Trace-ID wird automatisch weitergegeben, wenn das SDK verwendet wird. AWS

Wird verwendet`com.amazonaws.services.lambda.runtime.Context.getRemainingTimeInMillis()`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Wenn Sie virtuelle Threads in Ihrem Programm verwenden oder während der Initialisierung Threads erstellen, müssen Sie den erforderlichen Anforderungskontext an diese Threads übergeben.

Beispiel: Timeout-Behandlung

Prüfen Sie die verbleibende Zeit vor jeder Arbeitseinheit und beenden Sie die Verarbeitung, bevor der Timeout ausgelöst wird. Konfigurieren Sie die Konfiguration auf der BUFFER_MS Grundlage der erwarteten Dauer Ihres nächsten Arbeitsabschnitts.

```
private static final int BUFFER_MS = 2000; // Configure based on your next chunk of
 work

public Map<String, Object> handleRequest(Map<String, Object> event, Context context) {
    for (Object item : (List<Object>) event.get("items")) {
        if (context.getRemainingTimeInMillis() < BUFFER_MS)
            return Map.of("statusCode", 206, "body", "Timeout approaching, stopping
 early");
        processItem(item);
    }
    return Map.of("statusCode", 200, "body", "Done");
}
```

Beispiel: Weitergabe von Terminen an Downstream-Aufrufe

Wenn Sie nachgelagerte Dienste aufrufen, geben Sie die verbleibende Zeit als Timeout an, um zu vermeiden, dass Netzwerkaufrufe, die Ihren Aufruf überdauern würden, hängen bleiben. Verwenden Sie eine Überschreibung pro Anfrage auf einem gemeinsam genutzten Client, anstatt pro Aufruf einen neuen Client zu erstellen:

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.GetObjectRequest;
import java.time.Duration;

private static final S3Client s3 = S3Client.create();

public String handleRequest(Map<String, Object> event, Context context) {
    Duration timeout = Duration.ofMillis(Math.max(1000,
 context.getRemainingTimeInMillis() - 500));

    GetObjectRequest req = GetObjectRequest.builder()
        .bucket("my-bucket").key("my-key")
        .overrideConfiguration(cfg -> cfg.apiCallTimeout(timeout))
```

```
        .build();
    s3.getObject(req);
    return "Done";
}
```

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Ausführungsumgebung. Objekte, die während der Initialisierung erstellt wurden, werden von allen Threads gemeinsam genutzt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Sie können SIGTERM-Ereignisse abonnieren, um Aufgaben zur Funktionsbereinigung auszulösen, z. B. das Schließen von Datenbankverbindungen. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen, die abhängig sind

Für Lambda Managed Instances sind die folgenden Mindestpaketversionen erforderlich:

- AWS SDK für Java 2.0: Version 2.34.0 oder höher
- AWS X-Ray SDK für Java: Version 2.20.0 oder höher
- AWS Distribution für OpenTelemetry - Instrumentation für Java: Version 2.20.0 oder höher
- Powertools für AWS Lambda (Java): Version 2.8.0 oder höher

Powertools für AWS Lambda (Java)

Powertools for AWS Lambda (Java) ist mit Lambda Managed Instances kompatibel und bietet Hilfsprogramme für Logging, Tracing, Metriken und mehr. Weitere Informationen finden Sie unter Powertools for AWS Lambda (Java).

Nächste Schritte

- Überprüfen Sie die Node.js Laufzeit für von Lambda Managed Instances
- Überprüfen Sie die Python-Laufzeit für von Lambda verwaltete Instances
- Überprüfen Sie die .NET-Laufzeit für von Lambda verwaltete Instances
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

Node.js Laufzeit für Lambda Managed Instances

Für Node.js Laufzeiten verwendet Lambda Managed Instances Worker-Threads mit`async/await`-basierter Ausführung, um gleichzeitige Anfragen zu verarbeiten. Die Funktionsinitialisierung erfolgt einmal pro Worker-Thread. Gleichzeitige Aufrufe werden in zwei Dimensionen verarbeitet: Worker-Threads sorgen für Parallelität zwischen den vCPUs, und die asynchrone Ausführung sorgt für Parallelität innerhalb jedes Threads. Jede gleichzeitige Anforderung, die von demselben Worker-Thread bearbeitet wird, hat dasselbe Handler-Objekt und denselben globalen Status, sodass eine sichere Verarbeitung bei mehreren gleichzeitigen Anforderungen erforderlich ist.

Maximale Parallelität

Die maximale Anzahl gleichzeitiger Anfragen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für Node.js Laufzeiten ist die Standardeinstellung 64 gleichzeitige Anforderungen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Lambda passt die Anzahl der gleichzeitigen Anforderungen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität der einzelnen Ausführungsumgebungen, diese Anforderungen zu verarbeiten.

Denn die Anzahl der gleichzeitigen Anforderungen Node.js, die jede Ausführungsumgebung verarbeiten kann, wird durch die Anzahl der Worker-Threads und die Fähigkeit jedes Worker-Threads bestimmt, gleichzeitige Anforderungen asynchron zu verarbeiten. Die Standardanzahl der Worker-Threads wird durch die Anzahl der verfügbaren vCPUs bestimmt, oder Sie können die Anzahl der Worker-Threads konfigurieren, indem Sie die Umgebungsvariable festlegen. `AWS_LAMBDA_NODEJS_WORKER_COUNT` Wir empfehlen, asynchrone Funktionshandler zu verwenden, da auf diese Weise mehrere Anfragen pro Worker-Thread verarbeitet werden können. Wenn Ihr Funktionshandler synchron ist, kann jeder Worker-Thread jeweils nur eine einzige Anforderung verarbeiten.

Erstellung von Funktionen für Mehrfachparallelität

Mit einem asynchronen Funktionshandler verarbeitet jeder Runtime-Worker mehrere Anfragen gleichzeitig. Globale Objekte werden von mehreren gleichzeitigen Anforderungen gemeinsam genutzt. Vermeiden Sie bei veränderlichen Objekten die Verwendung von Global State oder use. `AsyncLocalStorage`

AWS SDK-Clients sind asynchron-sicher und erfordern keine besondere Behandlung.

Beispiel: Globaler Staat

Der folgende Code verwendet ein globales Objekt, das im Funktionshandler mutiert ist. Dies ist nicht asynchron-sicher.

```
let state = {
    currentUser: null,
    requestData: null
};

export const handler = async (event, context) => {
    state.currentUser = event.userId;
    state.requestData = event.data;

    await processData(state.requestData);

    // state.currentUser might now belong to a different request
    return { user: state.currentUser };
};
```

Durch die Initialisierung des `state` Objekts im Funktionshandler wird ein gemeinsamer globaler Status vermieden.

```
export const handler = async (event, context) => {
    let state = {
        currentUser: event.userId,
        requestData: event.data
    };

    await processData(state.requestData);

    return { user: state.currentUser };
};
```

Beispiel: Datenbankverbindungen

Der folgende Code verwendet ein gemeinsames Client-Objekt, das von mehreren Aufrufen gemeinsam genutzt wird. Je nach verwendeter Verbindungsbibliothek ist dies möglicherweise nicht sicher für Parallelität.

```
const { Client } = require('pg');

// Single connection created at init time
const client = new Client({
```

```
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD
});

// Connect once during cold start
client.connect();

exports.handler = async (event) => {
  // Multiple parallel invocations share this single connection = BAD
  // With multi-concurrent Lambda, queries will collide
  const result = await client.query('SELECT * FROM users WHERE id = $1',
 [event.userId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0])
  };
};
```

Ein parallelitätssicherer Ansatz besteht darin, einen Verbindungspool zu verwenden. Der Pool verwendet für jede gleichzeitige Datenbankabfrage eine separate Verbindung.

```
const { Pool } = require('pg');

// Connection pool created at init time
const pool = new Pool({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  max: 20,  // Max connections in pool
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
});

exports.handler = async (event) => {
  // Pool gives each parallel invocation its own connection
  const result = await pool.query('SELECT * FROM users WHERE id = $1', [event.userId]);

  return {
    statusCode: 200,
```

```
    body: JSON.stringify(result.rows[0])
  };
};
```

Node.js 22 Callback-basierte Handler

Wenn Sie Node.js 22 verwenden, können Sie keinen Callback-basierten Funktionshandler mit Lambda Managed Instances verwenden. Callback-based Handler werden nur für Lambda-Funktionen (Standard) unterstützt. Für Laufzeiten ab Node.js 24 Jahren sind Callback-basierte Funktionshandler sowohl für Lambda (Standard) als auch für Lambda Managed Instances veraltet.

Verwenden Sie stattdessen einen Funktionshandler, wenn Sie Lambda Managed Instances `async` verwenden. Weitere Informationen finden Sie unter Definieren Sie den Lambda-Funktionshandler in. Node.js

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anforderungen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn ein anderer Prozess die Datei überschreibt. Um dieses Problem zu beheben, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie pro Anfrage eindeutige Dateinamen, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz ausgeschöpft wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge aus verschiedenen Anfragen werden in Protokollen verschachtelt) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal. Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. ??? Dieses Format beinhaltet das`requestId`, sodass Logeinträge einer einzelnen Anfrage zugeordnet werden können. Wenn Sie den `console` Logger verwenden, `requestId` ist der automatisch in jedem Protokolleintrag enthalten. Weitere Informationen finden Sie unterthe section called “Verwenden von erweiterten Lambda-Protokollierungssteuerelementen mit Node.js”.

Beliebte Protokollierungsbibliotheken von Drittanbietern, wie Winston, unterstützen in der Regel die Verwendung der Konsole für die Protokollausgabe.

Kontext anfordern

Using `context.awsRequestId` bietet asynchronen Zugriff auf die Anforderungs-ID für die aktuelle Anfrage.

Wird für `context.xRayTraceId` den Zugriff auf die X-Ray Trace-ID verwendet. Dies ermöglicht den parallelen Zugriff auf die Trace-ID für die aktuelle Anfrage. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable mit Lambda Managed Instances nicht. Die X-Ray Trace-ID wird automatisch weitergegeben, wenn das SDK verwendet wird. AWS

Wird verwendet`context.getRemainingTimeInMillis()`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Beispiel: Behandlung von Timeouts

Prüfen Sie die verbleibende Zeit vor jeder Arbeitseinheit und beenden Sie die Verarbeitung, bevor der Timeout ausgelöst wird. Konfigurieren Sie die Konfiguration auf der `BUFFER_MS` Grundlage der erwarteten Dauer Ihres nächsten Arbeitsabschnitts.

```
const BUFFER_MS = 2000; // Configure based on your next chunk of work

exports.handler = async (event, context) => {
    for (const item of event.items) {
        if (context.getRemainingTimeInMillis() < BUFFER_MS)
            return { statusCode: 206, body: "Timeout approaching, stopping early" };
        await processItem(item);
    }
    return { statusCode: 200, body: "Done" };
};
```

Beispiel: Weitergabe von Terminen an Downstream-Aufrufe

Wenn Sie nachgelagerte Dienste aufrufen, geben Sie die verbleibende Zeit als Timeout an, um zu vermeiden, dass Netzwerkaufrufe, die Ihren Aufruf überdauern würden, hängen bleiben:

```
const { S3Client, GetObjectCommand } = require("@aws-sdk/client-s3");
const client = new S3Client({});

exports.handler = async (event, context) => {
    const timeout = Math.max(1000, context.getRemainingTimeInMillis() - 500);
    const response = await client.send(
        new GetObjectCommand({ Bucket: "my-bucket", Key: "my-key" }),
```

```
        { abortSignal: AbortSignal.timeout(timeout) }
    );
    return { statusCode: 200, body: "Done" };
};
```

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Worker-Thread. Möglicherweise werden wiederholte Protokolleinträge angezeigt, wenn Ihre Funktion während der Initialisierung Protokolle ausgibt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Lambda-Funktionen (Standard) mit Erweiterungen können das SIGTERM-Signal auch mithilfe von abonnieren. `process.on()` Dies wird für Funktionen, die Lambda Managed Instances verwenden, nicht unterstützt, da es `process.on()` nicht mit Worker-Threads verwendet werden kann. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen, die abhängig sind

Für Lambda Managed Instances sind die folgenden Mindestpaketversionen erforderlich:

- AWS SDK für JavaScript v3: Version 3.933.0 oder höher
- AWS X-Ray SDK für Node.js: Version 3.12.0 oder höher
- AWS Distro für OpenTelemetry - Instrumentierung für JavaScript: Version 0.8.0 oder höher
- Powertools for AWS Lambda (TypeScript): Version 2.29.0 oder höher

Elektrowerkzeuge für AWS Lambda () TypeScript

Powertools for AWS Lambda (TypeScript) ist mit Lambda Managed Instances kompatibel und bietet Hilfsprogramme für Logging, Tracing, Metriken und mehr. Weitere Informationen finden Sie unter Powertools for Lambda (). AWS TypeScript

Nächste Schritte

- Überprüfen Sie die Java-Laufzeit für von Lambda verwaltete Instances
- Überprüfen Sie die Python-Laufzeit für von Lambda verwaltete Instances
- Überprüfen Sie die .NET-Laufzeit für von Lambda verwaltete Instances
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

Python-Laufzeit für Lambda Managed Instances

Die Lambda-Laufzeit verwendet mehrere Python-Prozesse, um gleichzeitige Anfragen zu verarbeiten. Jede gleichzeitige Anforderung wird in einem separaten Prozess mit eigenem Speicherplatz und eigener Initialisierung ausgeführt. Jeder Prozess verarbeitet jeweils eine Anfrage synchron. Prozesse teilen sich den Speicher nicht direkt, sodass globale Variablen, Caches auf Modulebene und Singleton-Objekte zwischen gleichzeitigen Anforderungen isoliert werden.

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anforderungen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für Python-Laufzeiten ist die Standardeinstellung 16 gleichzeitige Anforderungen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Dieser Wert bestimmt auch die Anzahl der Prozesse, die von der Python-Laufzeit verwendet werden. Lambda passt die Anzahl der gleichzeitigen Anforderungen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität der einzelnen Ausführungsumgebungen, diese Anforderungen zu verarbeiten.

> **Important**
>
> Die Verwendung prozessbasierter Parallelität bedeutet, dass jeder Runtime-Worker-Prozess seine eigene Initialisierung durchführt. Die gesamte Speicherbelegung entspricht dem Arbeitsspeicher pro Prozess multipliziert mit der Anzahl der gleichzeitigen Prozesse. Wenn Sie große Bibliotheken oder Datensätze laden und eine hohe Parallelität haben, haben Sie einen großen Speicherbedarf. Je nach Arbeitslast müssen Sie möglicherweise Ihr CPU-to-memory Verhältnis anpassen oder eine niedrigere Einstellung für die Parallelität verwenden, um eine Überschreitung des verfügbaren Speichers zu vermeiden. Sie können die `MemoryUtilization` Metrik verwenden CloudWatch , um den Speicherverbrauch zu verfolgen.

Erstellung von Funktionen für Mehrfachparallelität

Aufgrund des prozessbasierten Mehrparallelitätsmodells greifen Lambda Managed Instances-Funktionen, die Python-Laufzeiten verwenden, bei mehreren Aufrufen nicht gleichzeitig auf speicherinterne Ressourcen zu. Sie müssen keine Codierungspraktiken anwenden, um die Sicherheit der Parallelität im Speicher zu gewährleisten.

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anforderungen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn ein anderer Prozess die Datei überschreibt. Um dieses Problem zu beheben, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Prozess oder pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz ausgeschöpft wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge aus verschiedenen Anfragen werden in Protokollen verschachtelt) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal.

Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. ??? Dieses Format beinhaltet das`requestId`, sodass Logeinträge einer einzelnen Anfrage zugeordnet werden können. Wenn Sie das `logging` Modul aus der Python-Standardbibliothek in Lambda verwenden, `requestId` ist das automatisch in jedem Protokolleintrag enthalten. Weitere Informationen finden Sie unter Verwenden der erweiterten Lambda-Logging-Steuerelemente mit Python.

Kontext anfordern

Wird verwendet`context.aws_request_id`, um auf die Anforderungs-ID für die aktuelle Anfrage zuzugreifen.

Bei Python-Laufzeiten können Sie die `_X_AMZN_TRACE_ID` Umgebungsvariable verwenden, um auf die X-Ray Trace-ID mit Lambda Managed Instances zuzugreifen. Die X-Ray Trace-ID wird automatisch weitergegeben, wenn Sie das SDK verwenden. AWS

Wird verwendet`context.get_remaining_time_in_millis()`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Beispiel: Behandlung von Timeouts

Prüfen Sie die verbleibende Zeit vor jeder Arbeitseinheit und beenden Sie die Verarbeitung, bevor der Timeout ausgelöst wird. Konfigurieren Sie die Konfiguration auf der `BUFFER_MS` Grundlage der erwarteten Dauer Ihres nächsten Arbeitsabschnitts.

```
BUFFER_MS = 2000  # Configure based on your next chunk of work

def handler(event, context):
    for item in event["items"]:
        if context.get_remaining_time_in_millis() < BUFFER_MS:
            return {"statusCode": 206, "body": "Timeout approaching, stopping early"}
        process_item(item)
    return {"statusCode": 200, "body": "Done"}
```

Beispiel: Weitergabe von Terminen an Downstream-Aufrufe

Wenn Sie nachgelagerte Dienste aufrufen, geben Sie die verbleibende Zeit als Timeout an, um zu vermeiden, dass Netzwerkaufrufe, die Ihren Aufruf überdauern würden, hängen bleiben. Das boto3 SDK unterstützt keine Timeouts pro Anfrage auf einem vorhandenen Client, daher müssen Sie einen Client mit dem gewünschten Timeout erstellen. Prüfen Sie bei Funktionen mit hohem Durchsatz, ob ein fester Timeout, der bei der Initialisierung konfiguriert wird, besser geeignet ist als die Client-Erstellung pro Anfrage.

```
import boto3
from botocore.config import Config

def handler(event, context):
    remaining = context.get_remaining_time_in_millis() / 1000
    timeout = max(1, remaining - 0.5)
    s3 = boto3.client("s3", config=Config(read_timeout=timeout,
 connect_timeout=timeout))
    response = s3.get_object(Bucket="my-bucket", Key="my-key")
    return {"statusCode": 200, "body": "Done"}
```

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Prozess. Möglicherweise werden wiederholte Protokolleinträge angezeigt, wenn Ihre Funktion während der Initialisierung Protokolle ausgibt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Sie können SIGTERM-Ereignisse abonnieren, um Aufgaben zur Funktionsbereinigung auszulösen, z. B. das Schließen von Datenbankverbindungen. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen, die abhängig sind

Für Lambda Managed Instances sind die folgenden Mindestpaketversionen erforderlich:

- Powertools für AWS Lambda (Python): Version 3.23.0 oder höher

Powertools für AWS Lambda (Python)

Powertools for AWS Lambda (Python) ist mit Lambda Managed Instances kompatibel und bietet Hilfsprogramme für Logging, Tracing, Metriken und mehr. Weitere Informationen finden Sie unter Powertools for AWS Lambda (Python).

Nächste Schritte

- Überprüfen Sie die Java-Laufzeit für von Lambda verwaltete Instances
- Überprüfen Sie die Node.js Laufzeit für Lambda Managed Instances
- Überprüfen Sie die .NET-Laufzeit für Lambda Managed Instances
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

.NET-Laufzeit für verwaltete Lambda-Instanzen

Für .NET-Laufzeiten verwenden Lambda Managed Instances einen einzigen .NET-Prozess pro Ausführungsumgebung. Mithilfe von.NET-Aufgaben werden mehrere Anfragen gleichzeitig verarbeitet.

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anfragen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für .NET-Laufzeiten ist die Standardeinstellung 32 gleichzeitige Anfragen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Lambda passt die Anzahl der gleichzeitigen Anfragen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität jeder Ausführungsumgebung, um diese Anfragen aufzunehmen.

Erstellung von Funktionen für Mehrfachparallelität

Sie sollten bei der Verwendung von Lambda Managed Instances dieselben Sicherheitspraktiken für Parallelität anwenden wie in jeder anderen Umgebung mit mehreren gleichzeitigen Vorgängen.

Da das Handler-Objekt von allen Tasks gemeinsam genutzt wird, muss jeder veränderbare Status Thread-sicher sein. Dazu gehören Sammlungen, Datenbankverbindungen und alle statischen Objekte, die während der Anforderungsverarbeitung geändert werden.

AWS SDK-Clients sind Thread-sicher und erfordern keine besondere Behandlung.

Beispiel: Datenbank-Verbindungspools

Der folgende Code verwendet ein statisches Datenbankverbindungsobjekt, das von mehreren gleichzeitigen Anfragen gemeinsam genutzt wird. Das `SqlConnection` Objekt ist nicht threadsicher.

```
public class DBQueryHandler
{
    // Single connection shared across threads - NOT SAFE
    private SqlConnection connection;

    public DBQueryHandler()
    {
        connection = new SqlConnection("your-connection-string-here");
        connection.Open();
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

Um dieses Problem zu lösen, verwenden Sie für jede Anfrage eine separate Verbindung, die aus einem Verbindungspool stammt. ADO.NET Anbieter wie unterstützen `Microsoft.Data.SqlClient` automatisch das Verbindungspooling, wenn das Verbindungsobjekt geöffnet wird.

```
public class DBQueryHandler
{
    public DBQueryHandler()
    {
```

```
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var connection = new SqlConnection("your-connection-string-here");
        connection.Open();
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

Beispiel: Sammlungen

Standard-.NET-Sammlungen sind nicht threadsicher:

```
public class Handler
{
    private static List<string> items = new List<string>();
    private static Dictionary<string, object> cache = new Dictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

Verwenden Sie aus Sicherheitsgründen Sammlungen aus dem `System.Collections.Concurrent` Namespace:

```
public class Handler
{
    private static ConcurrentBag<string> items = new ConcurrentBag<string>();
    private static ConcurrentDictionary<string, object> cache = new
 ConcurrentDictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
```

```
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anfragen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn eine andere Anforderung die Datei überschreibt. Um dieses Problem zu lösen, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz aufgebraucht wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge verschiedener Anfragen werden in Protokollen verschachtelt) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal. Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. Dieses Format beinhaltet die`requestId`, sodass Protokolleinträge mit einer einzigen Anfrage korreliert werden können. Wenn Sie das `context.Logger` Objekt zum Generieren von Protokollen verwenden, `requestId` wird es automatisch in jeden Protokolleintrag aufgenommen. Weitere Informationen finden Sie unterthe section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET”.

Kontext anfordern

Verwenden Sie die `context.AwsRequestId` Eigenschaft, um auf die Anforderungs-ID für die aktuelle Anfrage zuzugreifen.

Verwenden Sie die `context.TraceId` Eigenschaft, um auf die X-Ray Trace-ID zuzugreifen. Dies ermöglicht einen gleichzeitigen Zugriff auf die Trace-ID für die aktuelle Anfrage. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable bei Lambda Managed Instances nicht. Die X-Ray Trace-ID wird bei Verwendung des SDK automatisch weitergegeben. AWS

Wird verwendet`ILambdaContext.RemainingTime`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Beispiel: Behandlung von Timeouts

Prüfen Sie die verbleibende Zeit vor jeder Arbeitseinheit und beenden Sie die Verarbeitung, bevor das Timeout ausgelöst wird. Konfigurieren Sie den Puffer auf der Grundlage der erwarteten Dauer Ihres nächsten Arbeitsabschnitts.

```
private static readonly TimeSpan Buffer = TimeSpan.FromMilliseconds(2000);

public APIGatewayProxyResponse FunctionHandler(APIGatewayProxyRequest request,
 ILambdaContext context)
{
    var items = JsonSerializer.Deserialize<List<string>>(request.Body);
    foreach (var item in items)
    {
        if (context.RemainingTime < Buffer)
            return new APIGatewayProxyResponse { StatusCode = 206, Body = "Timeout
 approaching, stopping early" };
        ProcessItem(item);
    }
    return new APIGatewayProxyResponse { StatusCode = 200, Body = "Done" };
}
```

Beispiel: Weitergabe von Terminen an nachgelagerte Anrufe

Wenn Sie Anrufe an nachgelagerte Dienste tätigen, geben Sie die verbleibende Zeit als Timeout an, um zu vermeiden, dass Sie an Netzwerkanrufen hängen bleiben, die Ihren Aufruf überdauern würden. Verwenden Sie a `CancellationToken` mit einem gemeinsam genutzten Client, anstatt pro Aufruf einen neuen Client zu erstellen:

```
using Amazon.S3;

private static readonly IAmazonS3 s3 = new AmazonS3Client();

public async Task<string> FunctionHandler(APIGatewayProxyRequest request,
 ILambdaContext context)
{
    var timeout = context.RemainingTime - TimeSpan.FromMilliseconds(500);
    if (timeout < TimeSpan.FromSeconds(1)) timeout = TimeSpan.FromSeconds(1);

    using var cts = new CancellationTokenSource(timeout);
    await s3.GetObjectAsync("my-bucket", "my-key", cts.Token);
    return "Done";
```

```
}
```

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Ausführungsumgebung. Objekte, die während der Initialisierung erstellt wurden, werden von allen Anfragen gemeinsam genutzt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Sie können SIGTERM-Ereignisse abonnieren, um Aufgaben zur Funktionsbereinigung auszulösen, z. B. das Schließen von Datenbankverbindungen. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen von Abhängigkeiten

Lambda Managed Instances erfordert die folgenden Mindestpaketversionen:

- Amazon.Lambda.Core: Version 2.7.1 oder höher
- Amazon.Lambda.RuntimeSupport: Version 1.14.1 oder höher
- OpenTelemetry.Instrumentation.AWSLambda: Version 1.14.0 oder höher
- AWSXRayRecorder.Core: Version 2.16.0 oder höher
- AWSSDK.Core: Version 4.0.0.32 oder höher

Elektrowerkzeuge für AWS Lambda (.NET)

Powertools for AWS Lambda (.NET) und AWS Distro for OpenTelemetry - Instrumentation for unterstützen DotNet derzeit keine Lambda Managed Instances.

Nächste Schritte

- Überprüfen Sie die Java-Laufzeit für Lambda Managed Instances
- Node.js Laufzeit für Lambda Managed Instances überprüfen
- Python-Laufzeit für Lambda Managed Instances überprüfen
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

Rust-Unterstützung für Lambda Managed Instances

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anforderungen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert für Rust ist 8 gleichzeitige Anforderungen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Dieser Wert bestimmt die Anzahl der Tokio-Aufgaben, die von der Laufzeit ausgelöst werden, und ist für die gesamte Lebensdauer der Ausführungsumgebung statisch. Jeder Worker verarbeitet jeweils genau eine Inflight-Anfrage, ohne dass ein Multiplexing pro Worker erforderlich ist. Lambda passt die Anzahl der gleichzeitigen Anforderungen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität der einzelnen Ausführungsumgebungen, diese Anforderungen zu verarbeiten.

Erstellung von Funktionen für Mehrfachparallelität

Sie sollten bei der Verwendung von Lambda Managed Instances dieselben Thread-Sicherheitspraktiken anwenden wie in jeder anderen Multithread-Umgebung. Da das Handler-Objekt von allen Worker-Threads gemeinsam genutzt wird, muss jeder veränderbare Status threadsicher sein. Dazu gehören Sammlungen, Datenbankverbindungen und alle statischen Objekte, die während der Anforderungsverarbeitung geändert werden.

Um die gleichzeitige Bearbeitung von Anfragen zu ermöglichen, fügen Sie Ihrer `Cargo.toml` Datei das `concurrency-tokio` Feature-Flag hinzu.

```
[dependencies]
lambda_runtime = { version = "1", features = ["concurrency-tokio"] }
```

Der `lambda_runtime::run_concurrent(…)` Einstiegspunkt muss innerhalb einer Tokio-Laufzeit aufgerufen werden, was in der Regel durch das `#[tokio::main]` Attribut Ihrer Hauptfunktion bereitgestellt wird. Ihr Handler-Closure muss Clone + Send implementieren. Dadurch kann das Framework Ihren Handler sicher für mehrere asynchrone Aufgaben gemeinsam nutzen. Wenn diese Grenzen nicht eingehalten werden, wird Ihr Code nicht kompiliert.

Wenn Sie einen gemeinsamen Status für alle Aufrufe benötigen (ein Datenbankpool, eine Konfigurationsstruktur), packen Sie ihn in Arc ein und klonen Sie ihn `Arc` in jeden Aufruf.

Alle AWS SDK für Rust-Clients sind gleichzeitig sicher und erfordern keine besondere Behandlung.

Beispiel: AWS SDK-Client

Das folgende Beispiel verwendet einen S3-Client, um bei jedem Aufruf ein Objekt hochzuladen. Der Client wird direkt in die Closure geklont, ohne: `Arc`

```
let config = aws_config::load_defaults(BehaviorVersion::latest()).await;
let s3_client = aws_sdk_s3::Client::new(&config);

run_concurrent(service_fn(move |event: LambdaEvent<Request>| {
    let s3_client = s3_client.clone(); // cheap clone, no Arc needed
    async move {
        s3_client.put_object()
            .bucket(&event.payload.bucket)
            .key(&event.payload.key)
            .body(event.payload.body.into_bytes().into())
            .send()
            .await?;
        Ok(Response { message: "uploaded".into() })
    }
}))
.await
```

Beispiel: Datenbank-Verbindungspools

Wenn Ihr Handler Zugriff auf einen gemeinsamen Status wie einen Client und eine Konfiguration benötigt, packen Sie ihn in Arc ein und klonen Sie den `Arc` bei jedem Aufruf:

```
#[derive(Debug)]
struct AppState {
    dynamodb_client: DynamoDbClient,
    table_name: String,
    cache_ttl: Duration,
}

let config = aws_config::load_defaults(BehaviorVersion::latest()).await;
let state = Arc::new(AppState {
    dynamodb_client: DynamoDbClient::new(&config),
    table_name: std::env::var("TABLE_NAME").expect("TABLE_NAME must be set"),
    cache_ttl: Duration::from_secs(300),
});

run_concurrent(service_fn(move |event: LambdaEvent<Request>| {
    let state = state.clone();
```

```
    async move { handle(event, state).await }
}))
.await
```

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Aufrufen in derselben Ausführungsumgebung gemeinsam genutzt. Verwenden Sie eindeutige Dateinamen pro Aufruf (schließen Sie beispielsweise die Anforderungs-ID ein) oder implementieren Sie explizite Dateisperren, um Datenbeschädigungen zu vermeiden.

Protokollierung

Das Verschachteln von Protokollen (die Verschachtelung von Protokolleinträgen aus verschiedenen Anfragen in Protokollen) ist in Systemen mit mehreren parallelen Vorgängen normal. Funktionen, die Lambda Managed Instances verwenden, unterstützen das strukturierte JSON-Protokollformat mithilfe der fortschrittlichen Protokollierungssteuerungen von Lambda. ??? Dieses Format beinhaltet das`requestId`, sodass Logeinträge einer einzelnen Anfrage zugeordnet werden können. Weitere Informationen finden Sie unterthe section called “Implementierung der erweiterten Protokollierung mit der Tracing Crate”.

Kontext anfordern

Das `Context` Objekt wird direkt an jeden Handleraufruf übergeben. Wird verwendet`event.context.request_id`, um auf die Anforderungs-ID für die aktuelle Anfrage zuzugreifen.

Wird `event.context.xray_trace_id` für den Zugriff auf die X-Ray Trace-ID verwendet. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable mit Lambda Managed Instances nicht. Die X-Ray Trace-ID wird automatisch weitergegeben, wenn Sie das AWS SDK für Rust verwenden.

Wird verwendet`event.context.deadline`, um Timeouts zu erkennen — sie enthält die Frist für den Aufruf in Millisekunden.

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Ausführungsumgebung. Objekte, die während der Initialisierung erstellt wurden, werden von allen Anfragen gemeinsam genutzt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet,

um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. `lambda_runtime`bietet mit spawn_graceful_shutdown_handler () ein Hilfsprogramm, das die Konfiguration der ordnungsgemäßen Signalverarbeitung beim Herunterfahren vereinfacht. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called "Ausführungsumgebung".

Versionen, die abhängig sind

Für Lambda Managed Instances ist die folgende Mindestpaketversion erforderlich:

- `lambda_runtime`: Version 1.1.1 oder höher, mit aktivierter Funktion `concurrency-tokio`
- Die unterstützte Mindestversion von Rust (MSRV) ist 1.84.0.

Netzwerke für Lambda Managed Instances

Wenn Sie Funktionen von Lambda Managed Instances ausführen, müssen Sie die Netzwerkkonnektivität konfigurieren, damit Ihre Lambda Event Functions auf Ressourcen außerhalb der VPC zugreifen können. Dazu gehören AWS Dienste wie Amazon S3 und DynamoDB. Die Konnektivität wird auch für die Übertragung von Telemetriedaten an CloudWatch Logs und benötigt. X-Ray

Verbindungsoptionen

Es gibt drei Hauptansätze für die Konfiguration der VPC-Konnektivität, die jeweils unterschiedliche Kompromisse in Bezug auf Kosten, Sicherheit und Komplexität beinhalten.

Öffentliches Subnetz mit einem Internet-Gateway

Diese Option verwendet ein öffentliches Subnetz mit direktem Internetzugang über ein Internet-Gateway. Sie können zwischen IPv4- und IPv6-Konfigurationen wählen.

IPv4 mit Internet-Gateway

Um die IPv4-Konnektivität mit einem Internet-Gateway zu konfigurieren

1. Erstellen oder verwenden Sie ein vorhandenes öffentliches Subnetz mit einem IPv4-CIDR-Block.
2. Hinzufügen eines Internet-Gateways zu Ihrer VPC.
3. Aktualisieren Sie die Routing-Tabelle, um den `0.0.0.0/0` Verkehr an das Internet-Gateway weiterzuleiten.

4. Stellen Sie sicher, dass den Ressourcen öffentliche IPv4-Adressen oder Elastic IP-Adressen zugewiesen sind.
5. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Datenverkehr auf den erforderlichen Ports zugelassen wird.

Diese Konfiguration bietet bidirektionale Konnektivität und ermöglicht sowohl ausgehende Verbindungen von Ihren Funktionen als auch eingehende Verbindungen aus dem Internet.

IPv6 mit Internet-Gateway

Um die IPv6-Konnektivität mit einem Internet-Gateway zu konfigurieren

1. Aktivieren Sie IPv6 auf Ihrer VPC.
2. Erstellen oder verwenden Sie ein vorhandenes öffentliches Subnetz mit einem zugewiesenen IPv6-CIDR-Block.
3. Schließen Sie ein Internet-Gateway an Ihre VPC an (dasselbe Internet-Gateway kann sowohl IPv4 als auch IPv6 verarbeiten).
4. Aktualisieren Sie die Routing-Tabelle, um den `::/0` Verkehr an das Internet-Gateway weiterzuleiten.
5. Stellen Sie sicher, dass die AWS Dienste, auf die Sie zugreifen müssen, IPv6 in Ihrer Region unterstützen.
6. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Datenverkehr an den erforderlichen Ports zugelassen wird.

Diese Konfiguration bietet bidirektionale Konnektivität mithilfe von IPv6-Adressierung.

IPv6 mit Internet-Gateway nur für ausgehenden Datenverkehr

So konfigurieren Sie IPv6-Konnektivität mit einem Internet-Gateway nur für ausgehenden Datenverkehr

1. Aktivieren Sie IPv6 auf Ihrer VPC.
2. Erstellen oder verwenden Sie ein vorhandenes öffentliches Subnetz mit einem zugewiesenen IPv6-CIDR-Block.
3. Schließen Sie ein Internet-Gateway nur für ausgehenden Datenverkehr an Ihre VPC an.
4. Aktualisieren Sie die Routing-Tabelle, um den Datenverkehr an das Internet-Gateway weiterzuleiten, das `::/0` nur für ausgehenden Verkehr bestimmt ist.

5. Stellen Sie sicher, dass die AWS Dienste, auf die Sie zugreifen müssen, IPv6 in Ihrer Region unterstützen.
6. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Datenverkehr an den erforderlichen Ports zugelassen wird.

Diese Konfiguration bietet nur ausgehende Konnektivität, wodurch eingehende Verbindungen aus dem Internet verhindert werden, während Ihre Funktionen ausgehende Verbindungen initiieren können.

VPC-Endpunkte

VPC-Endpunkte ermöglichen es Ihnen, Ihre VPC privat mit unterstützten AWS Diensten zu verbinden, ohne dass ein Internet-Gateway, ein NAT-Gerät, eine VPN-Verbindung oder eine AWS Direct Connect-Verbindung erforderlich ist. Der Verkehr zwischen Ihrer VPC und dem AWS Service verlässt das Amazon-Netzwerk nicht.

So konfigurieren Sie VPC-Endpoints

1. Öffnen Sie die Amazon VPC-Konsole unter console.aws.amazon. com/vpc/.
2. Wählen Sie im Navigationsbereich Endpunkte aus.
3. Wählen Sie Endpunkt erstellen aus.
4. Wählen Sie bei Service category (Servicekategorie) die Option AWS services (-Services) aus.
5. Wählen Sie unter Service-Name den Service-Endpunkt aus, den Sie benötigen (z. B. `com.amazonaws.region.s3` für Amazon S3).
6. Wählen Sie im Feld VPC Ihre VPC aus.
7. Wählen Sie für Subnetze die Subnetze aus, in denen Sie Endpunkt-Netzwerkschnittstellen erstellen möchten. Wählen Sie Subnetze in mehreren Availability Zones, um eine hohe Verfügbarkeit zu erzielen.
8. Wählen Sie für Sicherheitsgruppen die Sicherheitsgruppen aus, die den Security groups (Endpunkt-Netzwerkschnittstellen) zugeordnet werden sollen. Die Sicherheitsgruppen müssen eingehenden Datenverkehr von der Sicherheitsgruppe Ihrer Funktion an den erforderlichen Ports zulassen.
9. Wählen Sie Endpunkt erstellen aus.

Wiederholen Sie diese Schritte für jeden AWS Dienst, auf den Ihre Funktionen zugreifen müssen.

Privates Subnetz mit NAT-Gateway

Diese Option verwendet ein NAT-Gateway, um den Internetzugang für Ressourcen in privaten Subnetzen bereitzustellen und gleichzeitig die Ressourcen privat zu halten.

Um ein privates Subnetz mit einem NAT-Gateway zu konfigurieren

1. Erstellen Sie ein öffentliches Subnetz (falls noch keines vorhanden ist) mit einem CIDR-Block.
2. Hinzufügen eines Internet-Gateways zu Ihrer VPC.
3. Erstellen Sie ein NAT-Gateway im öffentlichen Subnetz und weisen Sie eine Elastic IP-Adresse zu.
4. Aktualisieren Sie die Routing-Tabelle des öffentlichen Subnetzes, um eine Route hinzuzufügen: `0.0.0.0/0` → Internet-Gateway.
5. Erstellen oder verwenden Sie ein vorhandenes privates Subnetz mit einem CIDR-Block.
6. Aktualisieren Sie die Routentabelle des privaten Subnetzes, um eine Route hinzuzufügen: `0.0.0.0/0` → NAT-Gateway.
7. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Verkehr an den erforderlichen Ports zugelassen wird.

Stellen Sie für hohe Verfügbarkeit ein NAT-Gateway in jeder Availability Zone bereit und konfigurieren Sie Routing-Tabellen pro Availability Zone, um das lokale NAT-Gateway zu verwenden. Dadurch werden AZ-übergreifende Datenübertragungsgebühren vermieden und die Ausfallsicherheit verbessert.

Auswahl einer Konnektivitätsoption

Berücksichtigen Sie bei der Auswahl einer Konnektivitätsoption die folgenden Faktoren:

Öffentliches Subnetz mit Internet-Gateway

- Einfachste Konfiguration mit niedrigsten Kosten
- Geeignet für Entwicklungs- und Testumgebungen
- Ressourcen können eingehende Verbindungen aus dem Internet empfangen (aus Sicherheitsgründen)
- Unterstützt sowohl IPv4 als auch IPv6

VPC-Endpunkte

- Höchste Sicherheit, der Verkehr bleibt im Netzwerk AWS
- Geringere Latenz im Vergleich zum Internet-Routing
- Empfohlen für Produktionsumgebungen mit strengen Sicherheitsanforderungen
- Höhere Kosten pro Endpunkt, pro Availability Zone und pro verarbeitetem GB
- Für hohe Verfügbarkeit ist in jeder Availability Zone ein Endpunkt erforderlich

Privates Subnetz mit NAT-Gateway

- Ressourcen bleiben privat, wenn kein eingehender Internetzugang besteht
- Standardmuster für Unternehmensarchitekturen
- Unterstützt den gesamten IPv4-Internetverkehr
- Moderate Kosten mit Stunden- und Datenverarbeitungsgebühren für das NAT-Gateway
- Unterstützt nur IPv4

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java Node.js, und Python
- Sicherheit und Berechtigungen für Lambda Managed Instances verstehen

Überwachung von mit Lambda verwalteten Instances

Lambda bietet zwei Arten von Überwachungsdaten für Ihre Kapazitätsanbieter: Metriken und Systemprotokolle. Zusammen geben Ihnen diese Informationen Einblick in die Ressourcenauslastung, Ereignisse im Instanzlebenszyklus und betriebliche Probleme.

- Metriken — Lambda veröffentlicht automatisch CloudWatch Kennzahlen für Ihre Kapazitätsanbieter, damit Sie die Ressourcenauslastung überwachen, Kosten verfolgen und die Leistung optimieren können. Weitere Informationen finden Sie unter CloudWatch Metriken für Lambda Managed Instances.
- Systemprotokolle — Lambda sendet automatisch CloudWatch Systemprotokolle an Logs, in denen Ereignisse und Fehler im Instance-Lebenszyklus für Ihren Kapazitätsanbieter aufgezeichnet

werden. Weitere Informationen finden Sie unter Systemprotokolle des Kapazitätsanbieters für Lambda Managed Instances.

CloudWatch Metriken für Lambda Managed Instances

Sie können Lambda Managed Instances mithilfe CloudWatch von Metriken überwachen. Lambda veröffentlicht automatisch Metriken, mit denen CloudWatch Sie die Ressourcenauslastung überwachen, die Kosten verfolgen und die Leistung optimieren können.

Verfügbare Metriken

Lambda Managed Instances bietet Metriken auf zwei Ebenen: auf Ebene des Kapazitätsanbieters und auf Ebene der Ausführungsumgebung.

Metriken auf Ebene des Kapazitätsanbieters

Metriken auf Ebene des Kapazitätsanbieters bieten einen Überblick über die gesamte Ressourcenauslastung Ihrer Instances. Diese Metriken verwenden die folgenden Dimensionen:

- CapacityProviderName- Der Name Ihres Kapazitätsanbieters
- InstanceType- Der EC2-Instance-Typ

Die folgenden Metriken zur Ressourcenauslastung sind verfügbar:

- CPUUtilization — Der Prozentsatz der CPU-Auslastung aller Instances im Kapazitätsanbieter
- MemoryUtilization— Der Prozentsatz der Speicherauslastung aller Instances im Capacity Provider

Die folgenden Kapazitätskennzahlen sind verfügbar:

- vCPUAvailable — Die Menge an vCPUs, die auf Instances zur Zuweisung zur Verfügung steht (in Anzahl)
- MemoryAvailable— Die Menge an Arbeitsspeicher, die auf den Instanzen für die Zuweisung verfügbar ist (in Byte)
- vCPUAllocated — Die Menge der vCPU, die Instanzen für Ausführungsumgebungen zugewiesen wurde (in Anzahl)
- MemoryAllocated— Die Menge des Speichers, der Instanzen für Ausführungsumgebungen zugewiesen ist (in Byte)

Metriken auf Ebene der Ausführungsumgebung

Metriken auf Ebene der Ausführungsumgebung bieten Einblick in die Ressourcenauslastung und Parallelität einzelner Funktionen. Diese Metriken verwenden die folgenden Dimensionen:

- CapacityProviderName- Der Name Ihres Kapazitätsanbieters
- FunctionName- Der Name Ihrer Lambda-Funktion
- Ressource — Zeigen Sie die Metriken für eine bestimmte Version einer Funktion nach Ressource an.

> **Note**
>
> Für Lambda Managed Instances (LMI) unterstützt die `Resource` Dimension nur Funktionsversionen. Das Format ist `<FunctionName>:<FunctionVersion>`.

Verfügbare Metriken für die Ausführungsumgebung:

- ExecutionEnvironmentConcurrency- Die maximale Parallelität über einen Probenzeitraum von 1 Minute
- ExecutionEnvironmentConcurrencyLimit- Die maximale Parallelitätsgrenze pro Ausführungsumgebung
- ExecutionEnvironmentCPUUtilization- Der Prozentsatz der CPU-Auslastung für die Ausführungsumgebungen der Funktion
- ExecutionEnvironmentMemoryUtilization- Der Prozentsatz der Speicherauslastung für die Ausführungsumgebungen der Funktion

Drosseln Sie die Metriken für

Lambda Managed Instances gibt detaillierte Metriken für den Grund der Drosselung aus, die die Ressourcenbeschränkung identifizieren, die zu einer Drosselung geführt hat. Für jede Drosselung gibt Lambda genau eine der folgenden Untermetriken mit einem Wert von 1 aus, während die verbleibenden drei mit einem Wert von 0 ausgegeben werden:

- ConcurrencyThrottles— Die Ausführungsumgebung hat ihre maximale Parallelitätsgrenze erreicht. Dies kann behoben werden, indem die Ausführungsumgebungen entweder erhöht ExecutionEnvironmentMaxConcurrency oder aggressiver ??? skaliert werden.

- CPUThrottles — Die Ausführungsumgebung hat die zugewiesenen CPU-Ressourcen ausgeschöpft. Dies kann behoben werden, indem Sie die vCPU-Zuweisung Ihrer Funktion erhöhen oder indem Sie die Last pro Umgebung auf eine niedrigere Ebene reduzieren ExecutionEnvironmentMaxConcurrency.
- MemoryThrottles— Die Ausführungsumgebung hat den zugewiesenen Speicher ausgeschöpft. Dies kann behoben werden, indem Sie die Speicherzuweisung Ihrer Funktion erhöhen oder indem Sie die Last pro Umgebung ExecutionEnvironmentMaxConcurrency auf eine geringere Belastung reduzieren.
- DiskThrottles— Die Ausführungsumgebung hat den zugewiesenen Speicherplatz ausgeschöpft. Dieses Problem kann behoben werden, indem Sie den temporären Speicher Ihrer Funktion erhöhen oder indem Sie die Last pro Umgebung ExecutionEnvironmentMaxConcurrency auf ein niedrigeres Niveau reduzieren.

Lambda gibt neben diesen Untermetriken immer die `Throttles` Standardmetrik aus. Verwenden Sie diese Metriken, um zu verstehen, warum Ihre Funktion skaliert und ob Sie Ihre maximale Parallelität, den Funktionsspeicher, die vCPU-Zuweisung oder die Festplattenkonfiguration anpassen sollten. Die vollständige Liste der Lambda-Metriken finden Sie unter. Arten von Metriken für Lambda-Ereignisfunktionen

Häufigkeit und Aufbewahrung von Metriken

Die Metriken für Lambda Managed Instances werden in Intervallen von 1 Minute veröffentlicht und 15 Monate lang aufbewahrt.

Metriken anzeigen in CloudWatch

So zeigen Sie Lambda Managed Instances-Metriken in der Konsole an CloudWatch

1. Öffnen Sie die CloudWatch Konsole unter console.aws.amazon. com/cloudwatch/.
2. Wählen Sie im Navigationsbereich Metriken aus.
3. Wählen Sie auf der Registerkarte „Alle Metriken“ AWS/Lambda.
4. Wählen Sie die metrische Dimension aus, die Sie anzeigen möchten:
   - Filtern Sie für Metriken auf Ebene des Kapazitätsanbieters nach CapacityProviderName und InstanceType
   - Filtern Sie für Metriken auf der Ebene der Ausführungsumgebung nach CapacityProviderName FunctionName, und Ressource
5. Wählen Sie die Metriken aus, die Sie überwachen möchten.

Verwenden von Metriken zur Leistungsoptimierung

Überwachen Sie die CPU- und Speicherauslastung, um zu ermitteln, ob Ihre Funktionen richtig dimensioniert sind. Eine hohe Auslastung könnte auf den Bedarf an größeren Instance-Typen oder einer erhöhten Speicherzuweisung für Funktionen hinweisen. Verfolgen Sie Parallelitätsmetriken, um das Skalierungsverhalten zu verstehen und potenzielle Drosselungen zu identifizieren.

Überwachen Sie die Kapazitätskennzahlen, um sicherzustellen, dass ausreichend Ressourcen für Ihre Workloads verfügbar sind. Die Werte vCPUAvailable und MemoryAvailable Metriken helfen Ihnen dabei, die verbleibende Kapazität Ihrer Instances zu verstehen.

Nächste Schritte

- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances
- Lesen Sie die laufzeitspezifischen Anleitungen für Java und Python Node.js ???
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Machen Sie sich mit Sicherheit und Berechtigungen für Lambda Managed Instances vertraut

Systemprotokolle des Kapazitätsanbieters für Lambda Managed Instances

Um Ihnen bei der Überwachung und Fehlerbehebung in Ihrer Lambda Managed Instance-Umgebung zu helfen, generiert Lambda automatisch Systemprotokolle für Ihre Kapazitätsanbieter. Sie können Protokolle für Kapazitätsanbieter mithilfe der Lambda-Konsole, der CloudWatch Konsole, der AWS Command Line Interface (AWS CLI) oder der API anzeigen. CloudWatch Sie können Protokolle auch mithilfe von Logs-Abonnementfiltern an Amazon S3 und Firehose CloudWatch weiterleiten.

Solange die Betreiberrolle Ihres Kapazitätsanbieters über die erforderlichen Berechtigungen verfügt, erfasst Lambda Systemprotokolle und sendet sie an Amazon CloudWatch Logs, das das Standardziel ist.

> **Note**
>
> Die Systemprotokolle des Kapazitätsanbieters sind standardmäßig aktiviert. Lambda sendet CloudWatch Systemprotokolle automatisch an Logs, wenn Sie einen Kapazitätsanbieter erstellen.

- CloudWatch Logs ist das Standard-Protokollierungsziel für Systemprotokolle von Kapazitätsanbietern. CloudWatch Logs bietet Funktionen zur Anzeige und Analyse von Protokollen

in Echtzeit und unterstützt die Erstellung von Metriken und Alarmen auf der Grundlage Ihrer Protokolldaten.

- Amazon S3 ist wirtschaftlich für die langfristige Speicherung, und Services wie Athena können zur Analyse von Protokollen verwendet werden. Die Latenz ist in der Regel höher.
- Firehose bietet verwaltetes Streaming von Protokollen an verschiedene Ziele. Wenn Sie Protokolle an andere AWS Dienste (z. B. OpenSearch Service oder Redshift Data API) oder Drittanbieterplattformen (wie Datadog, New Relic oder Splunk) senden müssen, vereinfacht Firehose diesen Vorgang durch die Bereitstellung vorgefertigter Integrationen.

Konfigurieren von Protokollzielen

Lambda unterstützt mehrere Ziele für die Systemprotokolle Ihres Kapazitätsanbieters. Unabhängig von Ihrem ausgewählten Ziel bietet Lambda Optionen zur Steuerung der Protokollfilterung und -zustellung.

Die Systemprotokolle des Kapazitätsanbieters verwenden ausschließlich das strukturierte JSON-Format. Strukturierte JSON-Protokolle bieten eine verbesserte Durchsuchbarkeit und ermöglichen eine automatisierte Analyse. Sie können steuern, welche Protokolle Lambda an das von Ihnen gewählte Ziel sendet, indem Sie die Systemprotokollebene konfigurieren. Die Filterung hilft Ihnen, die Speicherkosten zu verwalten, und erleichtert das Auffinden relevanter Protokolleinträge beim Debuggen.

Weitere Informationen zum Einrichten der einzelnen Ziele finden Sie in den folgenden Themen:

- Systemprotokolle des Kapazitätsanbieters an CloudWatch Logs senden
- Systemprotokolle des Kapazitätsanbieters an Firehose senden
- Systemprotokolle des Kapazitätsanbieters an Amazon S3 senden

Konfiguration der Protokollierungssteuerungen für Kapazitätsanbieter

Um Ihnen mehr Kontrolle darüber zu geben, wie die Systemprotokolle Ihres Kapazitätsanbieters erfasst, verarbeitet und genutzt werden, bietet Lambda die folgenden Konfigurationsoptionen für die Protokollierung:

- Protokollebene — wählen Sie die Detailebene der Systemprotokolle, die Lambda an CloudWatch Logs sendet: `DEBUG`, `INFO`, oder. `WARN`

- Protokollgruppe — wählen Sie die Protokollgruppe CloudWatch Logs aus, an die Ihr Kapazitätsanbieter Logs sendet.

Weitere Informationen zur Konfiguration von Protokollierungssteuerungen finden Sie in den folgenden Themen:

- Log-level Filtern nach Systemprotokollen des Kapazitätsanbieters
- Systemprotokollformat des Kapazitätsanbieters
- Referenz zum Systemprotokoll des Kapazitätsanbieters

Systemprotokollformat des Kapazitätsanbieters

Die Systemprotokolle des Kapazitätsanbieters verwenden das strukturierte JSON-Format. Strukturierte JSON-Protokolle bieten eine verbesserte Durchsuchbarkeit und ermöglichen eine automatische Analyse der Betriebsereignisse Ihres Kapazitätsanbieters.

JSON-Format für Systemprotokolle

Jeder Systemprotokolleintrag wird als JSON-Objekt erfasst, das Schlüssel-Wert-Paare mit den folgenden Feldern auf oberster Ebene enthält:

Feld	Typ	Description
`timestamp`	Zeichenfolge	Die Uhrzeit, zu der das Protokollereignis generiert wurde, im ISO 8601-Format.
`level`	Zeichenfolge	Die Protokollebene des Ereignisses.
`message`	Zeichenfolge	Eine für Menschen lesbare Beschreibung des Ereignisses.
`metadata`	Objekt	Strukturierte Metadaten über das Ereignis. Enthält Felder, die je nach Ereignistyp variieren .

Metadaten-Felder

Das `metadata` Objekt enthält die folgenden Felder. Nicht alle Felder sind bei jedem Ereignis vorhanden.

Feld	Typ	Description
`instanceId`	Zeichenfolge	Die Amazon EC2-Instance-ID, die dem Ereignis zugeordnet ist.
`availabilityZones`	Zeichenfolgen-Array	Die Availability Zone (n), die dem Ereignis zugeordnet sind.
`instanceTypes`	Zeichenfolgen-Array	Die Amazon EC2-Instance-Typen, die dem Ereignis zugeordnet sind.
`capacityProviderArn`	Zeichenfolge	Der ARN des Kapazitätsanbieters.

Beispiel-Protokolleintrag

Das folgende Beispiel zeigt einen Systemprotokolleintrag für einen erfolgreichen Instance-Start:

```
{
  "timestamp": "2025-07-01T14:31:15.456Z",
  "level": "INFO",
  "message": "Successfully launched c7g.xlarge instance i-0abc123def456789 in us-
east-1a",
  "metadata": {
    "instanceId": "i-0abc123def456789",
    "availabilityZones": ["us-east-1a"],
    "instanceTypes": ["c7g.xlarge"],
    "capacityProviderArn": "arn:aws:lambda:us-east-1:123456789012:capacity-provider:my-
capacity-provider"
  }
}
```

Log-level Filtern nach Systemprotokollen des Kapazitätsanbieters

Lambda kann die Systemprotokolle Ihres Kapazitätsanbieters filtern, sodass nur Protokolle mit einer bestimmten Detailebene oder niedriger an CloudWatch Logs gesendet werden. Sie können zwischen den folgenden Protokollebenen wählen.

Protokollebene	Usage
DEBUG (am detailliertesten)	Detaillierte Informationen für das System-Debugging
INFO	Meldungen, die den normalen Betrieb des Kapazitätsanbieters aufzeichnen
WARN (am wenigsten Details)	Meldungen über potenzielle Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden

Wenn Sie eine Protokollebene auswählen, sendet Lambda Protokolle auf dieser Ebene und niedriger. Wenn Sie beispielsweise die Systemprotokollebene eines Kapazitätsanbieters auf INFO setzen, sendet Lambda keine Protokollausgaben auf der DEBUG-Ebene.

Standardmäßig setzt Lambda die Protokollebene des Systems auf INFO. Um detailliertere Protokolle zu erhalten, setzen Sie die Ebene auf DEBUG. Um weniger Protokolle zu erhalten, setzen Sie die Stufe auf WARN. Eine Liste der Protokollebenen, denen Lambda verschiedene Systemprotokollereignisse zuordnet, finden Sie unter Referenz zum Systemprotokoll des Kapazitätsanbieters.

Konfiguration der Systemprotokollebene

Um die Systemprotokollebene für Ihren Kapazitätsanbieter zu konfigurieren, können Sie die Lambda-Konsole oder die AWS Command Line Interface (AWS CLI) verwenden. Sie können die Systemprotokollebene auch mithilfe der `CreateCapacityProvider` `UpdateCapacityProvider` Lambda-API-Befehle konfigurieren.

So konfigurieren Sie die Systemprotokollebene (Konsole)

1. Öffnen Sie die Seite „Kapazitätsanbieter“ der Lambda-Konsole.
2. Wählen Sie einen Kapazitätsanbieter.
3. Wählen Sie Additional configuration (Zusätzliche Konfiguration).

4. Wählen Sie im Abschnitt Konfiguration der Protokollierung die Option Bearbeiten aus.
5. Wählen Sie das gewünschte Systemprotokoll-Level für Ihren Kapazitätsanbieter aus.
6. Wählen Sie Speichern.

So legen Sie die Systemprotokollebene bei der Erstellung eines Kapazitätsanbieters fest (AWS CLI)

- Verwenden Sie die `--telemetry-config` Option im Befehl create-capacity-provider, um die Systemprotokollebene zu konfigurieren, wenn Sie einen neuen Kapazitätsanbieter erstellen. Legen Sie `SystemLogLevel` auf `DEBUG`, `INFO` oder `WARN` fest.

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=subnet-12345,subnet-67890,SecurityGroupIds=sg-12345 \
  --permissions-config
 CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/MyOperatorRole \
  --telemetry-config LoggingConfig={SystemLogLevel=DEBUG}
```

Um die Systemprotokollebene eines vorhandenen Kapazitätsanbieters zu aktualisieren (AWS CLI)

- Verwenden Sie den Befehl update-capacity-provider, um die Systemprotokollebene eines vorhandenen Kapazitätsanbieters zu ändern.

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --telemetry-config LoggingConfig={SystemLogLevel=WARN}
```

Referenz zum Systemprotokoll des Kapazitätsanbieters

Lambda veröffentlicht Systemprotokollereignisse für den Lebenszyklus von Kapazitätsanbieter-Instances. Die folgende Tabelle zeigt die Ereignisse, die Lambda veröffentlicht, und zeigt, ob jedes Ereignis auf der Grundlage Ihrer konfigurierten Systemprotokollebene protokolliert wird.

Weitere Informationen zur Konfiguration der Systemprotokollebene finden Sie unterLog-level Filtern nach Systemprotokollen des Kapazitätsanbieters. Einzelheiten zum Format des Protokolleintrags finden Sie unterSystemprotokollformat des Kapazitätsanbieters.

Zuordnung auf Ereignisebene im Systemprotokoll

Veranstaltung	Description	**DEBUG**	**INFO**	**WARN**
Wird gestartet	Wird ausgegeben, wenn Lambda einen Instance-Start für den Kapazitätsanbieter initiiert.	Protokolliert	Protokolliert	Nicht protokolliert
Gestartet	Wird ausgegeben, wenn eine Instanz erfolgreich in einen aktiven Zustand übergeht und bereit ist, Aufrufe auszuführen.	Geloggt	Protokolliert	Nicht protokolliert
LaunchFailed	Wird ausgegeben, wenn ein Instanzstartversuch fehlschlägt.	Geloggt	Protokolliert	Protokolliert
Wird beendet	Wird ausgegeben, wenn Lambda eine Instanz zur Kündigung markiert.	Protokolliert	Protokolliert	Nicht protokolliert
Beendet	Wird ausgegeben, wenn eine Instanz erfolgreich beendet wurde.	Geloggt	Protokolliert	Nicht protokolliert
TerminateFailed	Wird ausgegeben, wenn Lambda eine Instance nicht beenden kann.	Geloggt	Protokolliert	Protokolliert
HealthCheckFailed	Wird ausgegeben, wenn eine Instanz ihre Integritätsprüfung nicht besteht und als fehlerhaft markiert wird.	Geloggt	Protokolliert	Protokolliert

> **Note**
>
> Diese Liste wird erweitert, da Lambda neue Funktionen für Kapazitätsanbieter hinzufügt.

Systemprotokolle des Kapazitätsanbieters an CloudWatch Logs senden

Standardmäßig erfasst Lambda automatisch Systemprotokolle für Ihren Kapazitätsanbieter und sendet sie an CloudWatch Logs, sofern die Betreiberrolle Ihres Kapazitätsanbieters über die erforderlichen Berechtigungen verfügt. Diese Protokolle werden in einer Protokollgruppe mit dem Namen `/aws/lambda/capacity-provider/<capacity-provider-name>` gespeichert. Bei Bedarf können Sie Ihren Kapazitätsanbieter so konfigurieren, dass er mithilfe der Lambda-Konsole oder der Lambda-API Protokolle an eine andere Protokollgruppe sendet. AWS CLI Weitere Informationen finden Sie unter Konfiguration von CloudWatch Protokollgruppen.

Sie können die Protokolle für Ihren Kapazitätsanbieter mithilfe der Lambda-Konsole, der CloudWatch Konsole, der AWS Command Line Interface (AWS CLI) oder der API anzeigen. CloudWatch Weitere Informationen finden Sie unter CloudWatch Protokolle für Kapazitätsanbieter anzeigen.

Erforderliche IAM-Berechtigungen

Die Betreiberrolle Ihres Kapazitätsanbieters benötigt die folgenden Berechtigungen, um CloudWatch Protokolle in Logs hochzuladen:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

Weitere Informationen finden Sie unter Verwenden identitätsbasierter Richtlinien (IAM-Richtlinien) für CloudWatch Protokolle im CloudWatch Amazon-Benutzerhandbuch.

Sie können diese CloudWatch Logs-Berechtigungen mithilfe der von Lambda bereitgestellten `AWSLambdaManagedEC2ResourceOperator` AWS verwalteten Richtlinie hinzufügen. Führen Sie den folgenden Befehl aus, um diese Richtlinie zu Ihrer Rolle hinzuzufügen:

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
AWSLambdaManagedEC2ResourceOperator
```

Weitere Informationen finden Sie unter the section called “Rolle des Operators”.

> **Important**
>
> Wenn die Operatorrolle nicht über die erforderlichen Berechtigungen verfügt, liefert Lambda keine Systemprotokolle und gibt keinen Fehler zurück. Stellen Sie sicher, dass Ihre Operator-

Rolle über die richtigen Berechtigungen verfügt, um Fehler bei der Protokollübermittlung im Hintergrund zu vermeiden.

Preisgestaltung

Für die Verwendung der Systemprotokolle des Kapazitätsanbieters fallen keine zusätzlichen Gebühren an. Es fallen jedoch die Standardgebühren für CloudWatch Protokolle an. Weitere Informationen finden Sie unter CloudWatch Preise.

Konfiguration von CloudWatch Protokollgruppen

Standardmäßig erstellt Lambda eine Protokollgruppe, die `/aws/lambda/capacity-provider/<capacity-provider-name>` nach Ihrem Kapazitätsanbieter benannt ist. Verwenden Sie die Lambda-Konsole oder die `UpdateCapacityProvider` API-Befehle `CreateCapacityProvider` und, um Protokolle an eine andere Protokollgruppe zu senden. AWS CLI

Wenn Sie eine benutzerdefinierte Protokollgruppe konfigurieren, muss der Name den Benennungsregeln für CloudWatch Logs entsprechen. Außerdem dürfen benutzerdefinierte Protokollgruppennamen nicht mit der Zeichenfolge beginnen`aws/`. Wenn Sie eine benutzerdefinierte Protokollgruppe erstellen, die mit beginnt`aws/`, kann Lambda die Protokollgruppe nicht erstellen, und Ihr Kapazitätsanbieter sendet keine Protokolle an CloudWatch.

Protokollstreams

Innerhalb der Protokollgruppe erstellt Lambda Log-Streams mit dem folgenden Benennungsformat:

```
<capacity-provider-name>/managed-instances
```

Für einen Kapazitätsanbieter mit dem Namen `my-capacity-provider` lautet der Log-Stream beispielsweise:

```
my-capacity-provider/managed-instances
```

Konfiguration einer benutzerdefinierten Protokollgruppe

So ändern Sie die Protokollgruppe eines Kapazitätsanbieters (Konsole)

1. Öffnen Sie die Seite „Kapazitätsanbieter“ der Lambda-Konsole.

2. Wählen Sie einen Kapazitätsanbieter.
3. Wählen Sie Additional configuration (Zusätzliche Konfiguration).
4. Wählen Sie im Abschnitt Konfiguration der Protokollierung die Option Bearbeiten aus.
5. Geben Sie den Namen der CloudWatch Logs-Log-Gruppe ein, an die Ihr Kapazitätsanbieter Logs senden soll. Wenn Sie den Namen einer vorhandenen Protokollgruppe eingeben, verwendet Ihr Kapazitätsanbieter diese Gruppe. Wenn keine Protokollgruppe mit dem von Ihnen eingegebenen Namen vorhanden ist, erstellt Lambda eine neue Protokollgruppe mit diesem Namen.
6. Wählen Sie Speichern.

Um die Protokollgruppe eines Kapazitätsanbieters zu ändern (AWS CLI)

- Verwenden Sie den Befehl update-capacity-provider, um die Protokollgruppe eines vorhandenen Kapazitätsanbieters zu ändern.

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --telemetry-config LoggingConfig={LogGroup=myLogGroup}
```

Um eine benutzerdefinierte Protokollgruppe anzugeben, wenn Sie einen Kapazitätsanbieter erstellen (AWS CLI)

- Um beim Erstellen eines neuen Kapazitätsanbieters eine benutzerdefinierte Protokollgruppe anzugeben, verwenden Sie die `--telemetry-config` Option im Befehl https://awscli.amazonaws.com/v2/documentation/api/latest/reference/lambda/create-capacity-provider.html create-capacity-provider.

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=subnet-12345,subnet-67890,SecurityGroupIds=sg-12345 \
  --permissions-config
 CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/MyOperatorRole \
  --telemetry-config LoggingConfig={LogGroup=myLogGroup}
```

Berechtigungen für Operatorrollen

Damit Ihr Kapazitätsanbieter CloudWatch Protokolle an Logs senden kann, muss die Operator-Rolle über die PutLogEvents Berechtigung logs: verfügen. Lambda fügt diese Berechtigung nicht automatisch hinzu. Wenn die Operator-Rolle sie noch nicht hat, fügen Sie sie hinzu, bevor Sie den Kapazitätsanbieter erstellen. Diese Berechtigung ist in der `AWSLambdaManagedEC2ResourceOperator` verwalteten Richtlinie enthalten.

Weitere Informationen zur Operatorrolle finden Sie unterthe section called “Rolle des Operators”.

CloudWatch Protokolle für Kapazitätsanbieter anzeigen

Sie können die Amazon CloudWatch Logs-Protokolle für Ihren Kapazitätsanbieter mithilfe der Lambda-Konsole, der CloudWatch Konsole oder der AWS Command Line Interface (AWS CLI) anzeigen. Folgen Sie den Anweisungen in den folgenden Abschnitten, um auf die Protokolle Ihres Kapazitätsanbieters zuzugreifen.

Streamen Sie CloudWatch Protokolle mit Logs Live Tail

Amazon CloudWatch Logs Live Tail hilft Ihnen dabei, Probleme bei Ihren Kapazitätsanbietern schnell zu beheben, indem es eine Streaming-Liste neuer Protokollereignisse in Echtzeit anzeigt. Sie können die aufgenommenen Logs von Ihrem Kapazitätsanbieter einsehen und filtern, sodass Sie Probleme schnell erkennen und lösen können.

> **Note**
>
> Bei Live Tail-Sitzungen fallen Kosten pro Minute an. Weitere Informationen zur Preisgestaltung finden Sie unter CloudWatch Amazon-Preise.

Um eine Live Tail-Sitzung zu starten, öffnen Sie die Protokollgruppe Ihres Kapazitätsanbieters in der CloudWatch Konsole und wählen Sie Live Tail. Weitere Informationen finden Sie im CloudWatch Amazon-Benutzerhandbuch unter Verwenden von Live Tail, um Protokolle nahezu in Echtzeit anzuzeigen.

Protokolle in der Lambda-Konsole anzeigen

Um Protokolle in der Lambda-Konsole anzuzeigen

1. Öffnen Sie die Seite „Kapazitätsanbieter“ der Lambda-Konsole.

2. Wählen Sie einen Kapazitätsanbieter.
3. Wählen Sie den Tab Überwachung.
4. Wählen Sie Anmeldungen anzeigen CloudWatch, um die Protokollgruppe in der CloudWatch Konsole zu öffnen.

Logs in der CloudWatch Konsole anzeigen

So zeigen Sie Protokolle in der CloudWatch Konsole an

1. Öffnen Sie die CloudWatch Konsole unter console.aws.amazon. com/cloudwatch/.
2. Wählen Sie im Navigationsbereich Protokolle und dann Protokollgruppen aus.
3. Wählen Sie die Protokollgruppe für Ihren Kapazitätsanbieter aus. Standardmäßig hat die Protokollgruppe einen Namen`/aws/lambda/capacity-provider/`*`<capacity-provider-name>`*.
4. Wählen Sie einen Protokollstream aus, um die Protokollereignisse anzuzeigen.

Sehen Sie sich Protokolle an mit dem AWS CLI

Verwenden Sie die `aws logs` Befehle AWS CLI, um Protokolle mit dem anzuzeigen. Im folgenden Beispiel werden die neuesten Protokollereignisse für einen Kapazitätsanbieter abgerufen:

```
aws logs get-log-events \
  --log-group-name /aws/lambda/capacity-provider/my-capacity-provider \
  --log-stream-name my-capacity-provider/managed-instances
```

Sie können CloudWatch Logs Insights auch verwenden, um die Systemprotokolle Ihres Kapazitätsanbieters abzufragen und zu analysieren. Im folgenden Beispiel werden alle Ereignisse der ERROR-Stufe abgefragt:

```
aws logs start-query \
  --log-group-name /aws/lambda/capacity-provider/my-capacity-provider \
  --start-time $(date -d '1 hour ago' +%s) \
  --end-time $(date +%s) \
  --query-string 'fields @timestamp, @message | filter level = "ERROR" | sort
 @timestamp desc'
```

Systemprotokolle des Kapazitätsanbieters an Firehose senden

Sie können Systemprotokolle des Kapazitätsanbieters an Amazon Data Firehose senden. Mit Firehose können Sie Protokolle in Echtzeit an verschiedene Ziele streamen, einschließlich Analysetools von Drittanbietern und benutzerdefinierte Endpunkte.

> **Note**
>
> Sie können die Systemprotokolle des Kapazitätsanbieters so konfigurieren, dass sie mithilfe der SDKs und aller SDKs an AWS CLI Firehose gesendet werden. AWS CloudFormation AWS

Preisgestaltung

Weitere Informationen zur Preisgestaltung finden Sie unter CloudWatch Amazon-Preise.

Einrichtung der Firehose-Protokollzustellung

Um Systemprotokolle des Kapazitätsanbieters an Firehose zu senden, erstellen Sie einen CloudWatch Logs-Abonnementfilter für die Protokollgruppe Ihres Kapazitätsanbieters. Der Abonnementfilter leitet Protokollereignisse an Ihren Firehose-Lieferstream weiter.

Standardmäßig hat die Protokollgruppe Ihres Kapazitätsanbieters einen Namen`/aws/lambda/capacity-provider/`*`<capacity-provider-name>`*. Verwenden Sie diese Protokollgruppe, wenn Sie den Abonnementfilter erstellen.

Anweisungen zur Einrichtung eines Abonnementfilters für Firehose, einschließlich der Erstellung von IAM-Rollen und Berechtigungen, finden Sie im CloudWatch Amazon-Benutzerhandbuch unter Abonnementfilter mit Firehose.

Cross-account Protokollierung

Sie können Ihren Kapazitätsanbieter so konfigurieren, dass er Protokolle an einen Firehose-Lieferstream in einem anderen AWS Konto sendet. Dies erfordert die Einrichtung eines Ziels und die Konfiguration der entsprechenden Berechtigungen in beiden Konten.

Anweisungen zur Einrichtung der kontoübergreifenden Protokollierung, einschließlich der erforderlichen IAM-Rollen und -Richtlinien, finden Sie im Amazon-Benutzerhandbuch unter Ein neues kontoübergreifendes Abonnement einrichten. CloudWatch

Systemprotokolle des Kapazitätsanbieters an Amazon S3 senden

Sie können Systemprotokolle des Kapazitätsanbieters an Amazon Simple Storage Service senden. Mit Amazon S3 können Sie Protokolle kostengünstig speichern, um sie langfristig aufzubewahren, und sie mithilfe von Diensten wie Athena analysieren.

> **Note**
>
> Mithilfe der SDKs und aller AWS SDKs können Sie die Systemprotokolle des Kapazitätsanbieters so konfigurieren AWS CLI AWS CloudFormation, dass sie an Amazon S3 gesendet werden.

Preisgestaltung

Weitere Informationen zur Preisgestaltung finden Sie unter CloudWatch Amazon-Preise.

Amazon S3-Protokollzustellung einrichten

Um Systemprotokolle des Kapazitätsanbieters an Amazon S3 zu senden, erstellen Sie einen CloudWatch Protokoll-Abonnementfilter für die Protokollgruppe Ihres Kapazitätsanbieters. Der Abonnementfilter leitet Protokollereignisse an Ihren Amazon S3-Bucket weiter.

Standardmäßig hat die Protokollgruppe Ihres Kapazitätsanbieters einen Namen`/aws/lambda/capacity-provider/`*`<capacity-provider-name>`*. Verwenden Sie diese Protokollgruppe, wenn Sie den Abonnementfilter erstellen.

Anweisungen zur Einrichtung eines Abonnementfilters für Amazon S3, einschließlich der Erstellung von IAM-Rollen und Bucket-Berechtigungen, finden Sie unterLambda Event Function-Protokolle an Amazon S3 senden.

Cross-account Protokollierung

Sie können Ihren Kapazitätsanbieter so konfigurieren, dass er Protokolle an einen Amazon S3-Bucket in einem anderen AWS Konto sendet. Dies erfordert die Einrichtung eines Ziels und die Konfiguration der entsprechenden Berechtigungen in beiden Konten.

Anweisungen zur Einrichtung der kontoübergreifenden Protokollierung, einschließlich der erforderlichen IAM-Rollen und -Richtlinien, finden Sie im CloudWatch Amazon-Benutzerhandbuch unter Ein neues kontoübergreifendes Abonnement einrichten.

Kontingente für Lambda Managed Instances

Auf dieser Seite werden die Dienstkontingente für AWS Lambda Managed Instances beschrieben. Diese Kontingente sind von AWS Lambda-Kontingenten (Standard) getrennt. Einige Kontingente können auf Anfrage erhöht werden.

Zusätzlich zu den auf dieser Seite aufgeführten Kontingenten unterliegen Lambda Managed Instances Amazon EC2-Servicekontingenten, die auf Ihr Konto angewendet werden. Zu den Kontingenten, von denen Lambda Managed Instance-Kunden häufig betroffen sind, gehören vCPU-Limits vom Typ EC2, Volumenlimits von Amazon EBS und verfügbare IP-Adressen in Ihren VPC-Subnetzen.

Lambda-API-Anforderungskontingente

Diese Kontingente steuern die Geschwindigkeit, mit der Sie API-Aufrufe tätigen können, um Kapazitätsanbieter für Lambda Managed Instances zu verwalten. Die API-Lese- und Schreibratenbegrenzungen gelten für alle Kapazitätsanbieteroperationen zusammen, einschließlich der Erstellung, Aktualisierung, Beschreibung und Löschung von Kapazitätsanbietern.

Ressource	Kontingent
Die maximale kombinierte Rate (Anfragen pro Sekunde) für alle Lese-APIs von Kapazitätsanbietern	15 Anforderungen pro Sekunde. Kann nicht erhöht werden.
Die maximale kombinierte Rate (Anfragen pro Sekunde) für alle Schreib-APIs von Kapazitätsanbietern	1 Anfrage pro Sekunde. Kann nicht erhöht werden.

Ressourcenkontingente für Lambda Managed Instances

Diese Kontingente definieren die Grenzwerte für die Kernressourcen von Lambda Managed Instances in Ihrem AWS Konto. Sie regeln die Anzahl der Kapazitätsanbieter, die Sie erstellen können, und die Anzahl der Funktionsversionen, die jedem Kapazitätsanbieter zugeordnet werden können.

Ressource	Kontingent
Kapazitätsanbieter	1.000. Die maximale Anzahl von Kapazitätsanbietern, die in einem Konto erstellt wurden.
Funktionsversionen pro Kapazitätsanbieter	100. Die maximale Anzahl von Funktionsversionen pro Kapazitätsanbieter. Kann nicht erhöht werden.
vCPUs pro Kapazitätsanbieter	15.000. Die maximale Anzahl von vCPUs pro Kapazität sanbieter.

Kontingente für die Zuordnung von Ereignisquellen

Diese Kontingente steuern den Durchsatz und die Konfigurationslimits für die Verarbeitung von Ereignissen verschiedener AWS Dienste auf Lambda Managed Instances. Die Durchsatzgrenzen sorgen für eine vorhersehbare Leistung, während die Grenzwerte für die Anzahl der Zuordnungen zur Aufrechterhaltung der Servicestabilität beitragen. Zuordnungen von Ereignisquellen auf Lambda Managed Instances unterstützen Amazon SQS, DynamoDB Streams, Amazon Kinesis Data Streams, Amazon MSK und selbstverwaltetes Apache Kafka als Ereignisquellen.

Ressource	Kontingent
Standarddurchsatz für die Zuordnung von SQS-Ereignisquellen auf Lambda Managed Instances	5 MB pro Sekunde. Kann nicht erhöht werden.
Standarddurchsatz für die Zuordnung von Kafka-Ereignisquellen auf Lambda Managed Instances	1 MB pro Sekunde. Kann nicht erhöht werden.
Standard-Kafka-Ereignisquellenzuordnungen auf Lambda Managed Instances	100 Zuordnungen von Ereignisquellen. Kann nicht erhöht werden.

Ressource	Kontingent
Durchsatz der Kinesis-Ereignisquellenzuordnung auf Lambda Managed Instances	25 MB pro Sekunde. Kann erhöht werden.
Durchsatz für die Zuordnung von DynamoDB-Ereignisquellen auf Lambda Managed Instances	10 MB pro Sekunde. Kann erhöht werden.
Rufen Sie den Anforderungsdurchsatz für asynchrone Aufrufe auf Lambda Managed Instances auf	5 MB pro Sekunde. Kann erhöht werden.

Beantragen einer Kontingenterhöhung

Für Kontingente, die erhöht werden können, können Sie eine Erhöhung über die Service Quotas Quotas-Konsole beantragen.

So fordern Sie eine Kontingenterhöhung an

1. Öffnen Sie die Service Quotas Quotas-Konsole unter console.aws.amazon. com/servicequotas/.
2. Wählen Sie im Navigationsbereich AWS -Services.
3. Wählen Sie AWS Lambda.
4. Wählen Sie das Kontingent aus, das Sie erhöhen möchten.
5. Wählen Sie Kontingenterhöhung anfordern.
6. Geben Sie den neuen Kontingentwert ein und begründen Sie die Erhöhung.
7. Wählen Sie Anfrage aus.

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java Node.js, und Python
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter

Bewährte Methoden für Lambda Managed Instances

Kapazitätsanbieter-Konfiguration

Separate Kapazitätsanbieter nach Vertrauensstufe. Erstellen Sie verschiedene Kapazitätsanbieter für Workloads mit unterschiedlichen Sicherheitsanforderungen. Alle Lambda-Ereignisfunktionen, die demselben Kapazitätsanbieter zugewiesen sind, müssen gegenseitig vertrauenswürdig sein, da Kapazitätsanbieter als Sicherheitsgrenze dienen.

Verwenden Sie aussagekräftige Namen. Benennen Sie Kapazitätsanbieter, um deren Verwendungszweck und Vertrauensstufe eindeutig anzugeben (z. B.`production-trusted`,`dev-sandbox`). Dies hilft den Teams, den Zweck und die Sicherheitslage der einzelnen Kapazitätsanbieter zu verstehen.

Verwenden Sie mehrere Availability Zones. Geben Sie bei der Erstellung von Kapazitätsanbietern Subnetze für mehrere Availability Zones an. Lambda startet aus Gründen der AZ-Resilienz standardmäßig drei Instances und gewährleistet so eine hohe Verfügbarkeit Ihrer Funktionen.

Auswahl des Instance-Typs

Lassen Sie Lambda die Instanztypen auswählen. Standardmäßig wählt Lambda die besten Instance-Typen für Ihren Workload aus. Wir empfehlen, Lambda Managed Instances die Instanztypen für Sie auswählen zu lassen, da eine Einschränkung der Anzahl möglicher Instanztypen zu einer geringeren Verfügbarkeit führen kann.

Geben Sie Instanztypen für bestimmte Anforderungen an. Wenn Sie bestimmte Hardwareanforderungen haben, legen Sie für zulässige Instanztypen eine Liste kompatibler Instanzen fest. Beispiel:

- Wählen Sie für Anwendungen, die eine hohe Netzwerkbandbreite erfordern, mehrere n-Instance-Typen aus
- Wählen Sie für Test- oder Entwicklungsumgebungen mit Kostenbeschränkungen kleinere Instance-Typen wie m7a.large

Konfiguration der Funktionen

Wählen Sie die entsprechenden Speicher- und vCPU-Einstellungen. Wählen Sie Speicher- und vCPU-Konfigurationen aus, die mehrere gleichzeitige Ausführungen Ihrer Funktion unterstützen. Die unterstützte Mindestfunktionsgröße beträgt 2 GB und 1 vCPU.

- Wählen Sie für Python-Anwendungen ein höheres Verhältnis von Arbeitsspeicher zu vCPUs (z. B. 4 zu 1 oder 8 zu 1), da Python Multiparallelität so handhabt
- Wählen Sie für CPU-intensive Operationen oder Funktionen, die wenig I/O ausführen, mehr als eine vCPU
- Bei IO-heavy Anwendungen wie Webdiensten oder Batch-Jobs bietet die Mehrfachparallelität den größten Vorteil

Konfigurieren Sie die maximale Parallelität entsprechend. Lambda wählt vernünftige Standardwerte für maximale Parallelität, die den Ressourcenverbrauch und den Durchsatz in Einklang bringen. Passen Sie diese Einstellung an die Ressourcennutzung Ihrer Funktion an:

- Erhöhen Sie die maximale Parallelität (bis zu 64 pro vCPU), wenn Ihre Funktionsaufrufe nur sehr wenig CPU verbrauchen
- Verringern Sie die maximale Parallelität, wenn Ihre Anwendung viel Speicher und sehr wenig CPU verbraucht

Beachten Sie, dass in Ausführungsumgebungen mit sehr geringer Parallelität Drosselungen und Skalierungsschwierigkeiten auftreten können.

Long-running Funktionen

Funktionen auf AWS Lambda verwalteten Instances können bis zu 90 Minuten (5.400 Sekunden) pro Aufruf für asynchrone Aufrufe und für Aufrufe von Ereignisquellenzuordnungen ausgeführt werden, mit Ausnahme von Amazon MQ- und Amazon DocumentDB DocumentDB-Ereignisquellenzuordnungen, die auf 15 Minuten begrenzt bleiben. Synchrone Aufrufe und die Phase der Funktionsinitialisierung bleiben ebenfalls auf 15 Minuten begrenzt. Weitere Informationen zum Einstellen des Timeouts finden Sie unter Lambda-Funktions-Timeout konfigurieren. Da Ihre Funktion länger ausgeführt werden kann, sollten Sie die folgenden Überlegungen zu kurzlebigen Komponenten wie Netzwerkverbindungen und Anmeldeinformationen beachten.

Berücksichtigen Sie Verbindungs-Timeouts im Leerlauf. Stellen Sie sicher, dass Verbindungstimeouts im Leerlauf bei Downstream-Diensten wie Amazon RDS ElastiCache, Amazon und externen APIs die volle Funktionsdauer abdecken. Wenn Ihre Funktion den Datenverkehr über ein NAT-Gateway weiterleitet, senden Sie Keep-Alive-Pakete, um zu verhindern, dass inaktive Verbindungen nach dem 350-Sekunden-Timeout des NAT-Gateways unterbrochen werden.

Respektieren Sie die DNS-TTL-Werte. Beachten Sie die DNS-TTL-Werte, wenn Sie externe Hostnamen auflösen. Die AWS SDKs behandeln dies automatisch, aber benutzerdefinierte HTTP-Clients können DNS-Auflösungen zwischenspeichern, die über ihre TTL hinausgehen.

Temporäre Anmeldeinformationen aktualisieren. Wenn Ihre Funktion temporäre Anmeldeinformationen oder Token abruft, stellen Sie sicher, dass sie für die gesamte Ausführungsdauer gültig bleiben, oder aktualisieren Sie sie im Hintergrund.

Design für Idempotenz. Lambda garantiert nicht, dass die Verarbeitung exakt einmal erfolgt. Je länger Funktionen ausgeführt werden, desto größer wird das Zeitfenster für Wiederholungsversuche und doppelte Lieferungen. Verwenden Sie Powertools for AWS Lambda, um Idempotenz für Operationen wie Zahlungen oder Datenbankschreibvorgänge zu implementieren, sodass sie dasselbe Ergebnis liefern, auch wenn sie mehr als einmal ausgeführt werden. Wenn Sie dauerhafte Lambda-Funktionen verwenden, haben Schritte eine Semantik, die mindestens einmal ausgeführt wird: Das Durable Execution SDK überspringt abgeschlossene Schritte während der Wiedergabe, aber Schritte, die vor dem Checkpointing fehlschlagen, können erneut ausgeführt werden. Sie können Ausführungsnamen als Idempotenzschlüssel verwenden.

Passen Sie die Zuordnungen der Ereignisquellen für eine längere Verarbeitung an. Legen Sie für Amazon SQS das Sichtbarkeits-Timeout der Warteschlange auf mindestens das Sechsfache des Funktions-Timeouts fest, sodass Lambda genügend Zeit hat, einen Batch erneut zu versuchen, wenn die Funktion gedrosselt wird. Lambda überprüft dies, wenn Sie die Ereignisquellenzuordnung erstellen, verhindert jedoch nicht, dass spätere Änderungen an der Warteschlange oder Funktion, die zu einer Nichtübereinstimmung führen, vorgenommen werden. Konfigurieren Sie für Amazon Kinesis Data Streams und Amazon DynamoDB Streams das maximale Batching-Fenster und den Parallelisierungsfaktor, um längere Verarbeitungszeiten pro Stapel zu berücksichtigen, und aktivieren Sie die Meldung teilweiser Batch-Fehler, sodass nur fehlgeschlagene Datensätze wiederholt werden und nicht der gesamte Stapel.

Skalierungskonfiguration

Legen Sie die entsprechende Zielressourcenauslastung fest. Standardmäßig bietet Lambda genügend Spielraum, sodass sich Ihr Traffic ohne Drosselungen innerhalb von 5 Minuten verdoppeln kann. Passen Sie dies an Ihre Workload-Merkmale an:

- Bei sehr gleichbleibenden Workloads oder Anwendungen, die nicht empfindlich auf Drosselungen reagieren, sollten Sie das Ziel auf einen hohen Wert setzen, um eine höhere Auslastung und niedrigere Kosten zu erreichen

- Setzen Sie sich bei Workloads mit potenziellen Datenverkehrsspitzen niedrige Ressourcenziele, um zusätzlichen Spielraum zu haben

Planen Sie das Verkehrswachstum ein. Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, wenn Lambda Instances und Ausführungsumgebungen skaliert. Entwerfen Sie Ihre Anwendung so, dass sie potenziellen Drosselungen in Zeiten schneller Skalierung begegnet.

Sicherheit

Wenden Sie die geringsten Rechte für Berechtigungen an. PassCapacityProvider Erteilen `lambda:PassCapacityProvider` Sie Berechtigungen nur für die erforderlichen Kapazitätsanbieter. Verwenden Sie Berechtigungen auf Ressourcenebene, um einzuschränken, welche Kapazitätsanbieter Benutzer Funktionen zuweisen können.

Überwachen Sie die Nutzung von Kapazitätsanbietern. Wird verwendet AWS CloudTrail , um die Zuweisungen und Zugriffsmuster von Kapazitätsanbietern zu überwachen. Dies hilft bei der Identifizierung unberechtigter Zugriffsversuche und gewährleistet die Einhaltung von Sicherheitsrichtlinien.

Trennen Sie nicht vertrauenswürdige Workloads. Verlassen Sie sich bei der Sicherheitsisolierung zwischen nicht vertrauenswürdigen Workloads nicht auf Container. Verwenden Sie verschiedene Kapazitätsanbieter, um Workloads zu trennen, denen nicht gegenseitig vertraut wird.

Kostenoptimierung

Verwenden Sie die EC2-Preisoptionen. Nutzen Sie EC2-Sparpläne und Reserved Instances, um die Kosten zu senken. Diese Preisoptionen gelten für die zugrunde liegende EC2-Rechenleistung (die Verwaltungsgebühr von 15% wird nicht rabattiert).

Optimieren Sie für stationäre Workloads. Lambda Managed Instances eignen sich am besten für Steady-State-Funktionen mit vorhersehbarem hohem Verkehrsaufkommen. Bei stark frequentierten Datenverkehrsmustern ist Lambda (Standard) möglicherweise kostengünstiger.

Überwachen Sie die Ressourcennutzung. Verfolgen Sie CloudWatch Metriken, um die CPU- und Speicherauslastung zu verstehen. Passen Sie die Funktionsspeicherzuweisung und die Auswahl des Instanztyps an die tatsächlichen Nutzungsmuster an, um die Kosten zu optimieren.

Überwachung und Beobachtbarkeit

Überwachen Sie die Kennzahlen der Kapazitätsanbieter. Verfolgen Sie Kennzahlen auf Ebene der Kapazitätsanbieter, einschließlich CPU-Auslastung MemoryUtilization, vCPUAvailable, und überprüfen Sie, ob ausreichend Ressourcen MemoryAvailable für Ihre Workloads verfügbar sind.

Überwachen Sie die Kennzahlen der Ausführungsumgebung. Verfolgen Sie Metriken auf der Ebene der Ausführungsumgebung, einschließlich ExecutionEnvironmentConcurrency und ExecutionEnvironmentConcurrencyLimit um das Skalierungsverhalten zu verstehen und mögliche Drosselungen zu identifizieren.

Richten Sie Alarme ein. CloudWatch Erstellen Sie CloudWatch Alarme für wichtige Kennzahlen, um Probleme proaktiv zu identifizieren:

- Hohe CPU- oder Speicherauslastung
- Niedrige verfügbare Kapazität
- Wir nähern uns den Grenzwerten für Parallelität

Language-specific Überlegungen

Folgen Sie den sprachspezifischen Best Practices. Jede Programmiersprache behandelt Multiparallelität unterschiedlich. Ausführliche Empfehlungen finden Sie in den sprachspezifischen Leitfäden:

- Java: Verwenden Sie Thread-sichere Sammlungen und für anforderungsspezifische `AtomicInteger` Status `ThreadLocal`
- Node.js: Verwenden Sie es InvokeStore für alle anforderungsspezifischen Status und vermeiden Sie globale Variablen
- Python: Verwenden Sie eindeutige Dateinamen `/tmp` mit Anforderungs-IDs und ziehen Sie eine prozessbasierte Speicherisolierung in Betracht
- Rust: Verwenden Sie `run_concurrent` stattrun, wenn die `concurrency-tokio` Funktion aktiviert ist. Der Handler muss `Clone` + sein`Send`.

Testen Sie auf Thread-Sicherheits- und Parallelitätsprobleme. Testen Sie Ihre Funktionen vor der Bereitstellung in der Produktion gründlich auf Thread-Sicherheitsprobleme, Race-Bedingungen und die korrekte Zustandsisolierung bei gleichzeitiger Last.

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java Node.js, und Python
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Überwachen Sie Lambda Managed Instances mit Metriken CloudWatch

Problembehandlung bei Lambda Managed Instances

Probleme mit der Drosselung und Skalierung

Gelegentliche Drosselung

Problem: Während des normalen Betriebs treten Drosselungsfehler (HTTP 429) auf.

Ursache: Lambda Managed Instances lehnen möglicherweise neue Aufrufe ab, um bereits ausgeführte Aufrufe zu schützen. Wenn Ihre Ausführungsumgebungen konstant stark ausgelastet sind, werden neue Aufrufe möglicherweise gedrosselt.

Lösung:

- Überwachen Sie die Skalierungsmetriken: Sehen Sie sich das Diagramm mit den Gründen der Drosselung an, um die Ursache für Probleme mit der Drosselung und Kapazitätsskalierung zu verstehen. Im folgenden Beispiel führt eine hohe CPU-Auslastung zu Drosselungen.
- Überprüfen Sie die Funktionskonfiguration: Stellen Sie sicher, dass Ihre Funktionsspeicher- und vCPU-Einstellungen mehrere gleichzeitige Ausführungen unterstützen. Erhöhen Sie bei Bedarf die Funktionsspeicher- oder vCPU-Zuweisung. Wenn `ExecutionEnvironmentVCPUUtilization` der Wert hoch ist, versuchen Sie, mehr vCPU pro Funktion hinzuzufügen.

  ```
  aws lambda update-function \
    --function-name my-function \
    ...
    --memory-size 8192
  ```

  Wenn `ExecutionEnvironmentMemoryUtilization` der Wert hoch ist, versuchen Sie, mehr Arbeitsspeicher pro vCPU hinzuzufügen:

```
aws lambda update-function \
  --function-name my-function \
  --capacity-provider-config '{
    "LambdaManagedInstancesCapacityProviderConfig": {
      "ExecutionEnvironmentMemoryGiBPerVCpu": 4.0
    }
  }'
```

Drosselt beim Hochskalieren

Problem: Drosselungsfehler (HTTP 429) treten auf, wenn der Datenverkehr schnell zunimmt.

Ursache: Lambda Managed Instances werden asynchron basierend auf der CPU-Ressourcenauslastung und der Auslastung mehrerer Parallelitäten skaliert. Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, da Lambda Instances und Ausführungsumgebungen entsprechend der Nachfrage hochskaliert.

Lösungen:

- Passen Sie die Zielressourcenauslastung an: Wenn Ihre Arbeitslast vorhersehbare Datenverkehrsmuster aufweist, legen Sie eine niedrigere Zielressourcenauslastung fest, um zusätzlichen Spielraum für Datenverkehrsspitzen zu haben.

```
aws lambda create-capacity-provider \
  --name my-capacity-provider \
  ...
  --capacity-provider-scaling-config '{
    "ScalingMode": "Manual",
    "ScalingPolicies": [
      {
        "PredefinedMetricType": "LambdaCapacityProviderAverageCPUUtilization",
        "TargetValue": 30.0
      }
    ]
  }'
```

- Pre-warm Kapazität: Verwenden Sie bei geplanten Datenverkehrszuwächsen die `PutFunctionScalingConfig` API, um zusätzliche Kapazitäten vorzuwärmen.

```
aws lambda put-function-scaling-config \
```

```
  --function-name my-function \
  --qualifier 1 \
  --function-scaling-config '{
    "MinExecutionEnvironments": 100
  }'
```

Langsames Herunterskalieren

Problem: Es dauert lange, bis Instances herunterskaliert sind, nachdem der Traffic abnimmt.

Ursache: Lambda Managed Instances werden schrittweise herunterskaliert, um die Verfügbarkeit aufrechtzuerhalten und schnelle Kapazitätsänderungen zu vermeiden, die sich auf die Leistung auswirken könnten.

Lösung:

Dieses Verhalten wird erwartet. Lambda skaliert Instances konservativ herunter, um Stabilität zu gewährleisten. Überwachen Sie Ihre CloudWatch Metriken, um die Anzahl der laufenden Instances zu verfolgen.

Probleme mit der Parallelität

Bei Ausführungsumgebungen mit geringer Parallelität kommt es zu Drosselungen

Problem: Ihre Funktionen werden gedrosselt, obwohl Kapazität verfügbar ist.

Ursache: In Ausführungsumgebungen mit sehr niedriger maximaler Parallelität kann es zu Schwierigkeiten bei der effektiven Skalierung kommen. Lambda Managed Instances sind für Anwendungen konzipiert, die mehrere Anwendungen gleichzeitig ausführen.

Lösungen:

- Erhöhen Sie die maximale Parallelität: Wenn Ihre Funktionsaufrufe sehr wenig CPU beanspruchen, erhöhen Sie die maximale Parallelitätseinstellung auf bis zu 64 pro vCPU.

```
aws lambda update-function \
  --function-name ordering-api-backend \
  --capacity-provider-config '{
    "LambdaManagedInstancesCapacityProviderConfig": {
      "PerExecutionEnvironmentMaxConcurrency": 32
```

```
    }
}'
```

- Funktionscode optimieren: Überprüfen Sie Ihren Funktionscode, um den CPU-Verbrauch pro Aufruf zu reduzieren und so eine höhere Parallelität zu ermöglichen.
- Passen Sie den Funktionsspeicher und die vCPU an: Stellen Sie sicher, dass Ihre Funktion über genügend Ressourcen verfügt, um mehrere gleichzeitige Aufrufe zu verarbeiten.

Probleme mit der Thread-Sicherheit (Java-Laufzeit)

Problem: Ihre Java-Funktion liefert falsche Ergebnisse oder es kommt unter Last zu Rennbedingungen.

Ursache: Mehrere Threads führen die Handler-Methode gleichzeitig aus, und Shared State ist nicht threadsicher.

Lösung:

- Verwenden Sie `AtomicInteger` oder `AtomicLong` für Zähler anstelle von primitiven Typen
- Ersetze `HashMap` durch `ConcurrentHashMap`
- Zum `Collections.synchronizedList()` Wickeln verwenden `ArrayList`
- Wird `ThreadLocal` für einen anforderungsspezifischen Status verwendet
- Greifen Sie über das Lambda-Context-Objekt auf Trace-IDs zu, nicht auf Umgebungsvariablen

Eine ausführliche Anleitung finden Sie in der Dokumentation zur Java-Laufzeit für Lambda Managed Instances.

Probleme mit der Zustandsisolierung (Node.js Laufzeit)

Problem: Ihre Node.js Funktion gibt Daten aus verschiedenen Anfragen zurück oder es kommt zu einer Datenbeschädigung.

Ursache: Globale Variablen werden bei gleichzeitigen Aufrufen im selben Worker-Thread gemeinsam genutzt. Wenn asynchrone Operationen die Kontrolle übernehmen, können andere Aufrufe den gemeinsamen Status ändern.

Lösung:

- Installation und Verwendung `@aws/lambda-invoke-store` für alle anforderungsspezifischen Zustände
- Ersetzen Sie globale Variablen durch und `InvokeStore.set()` `InvokeStore.get()`
- Verwenden Sie eindeutige Dateinamen `/tmp` mit Anforderungs-IDs
- Greifen Sie auf Trace-IDs `InvokeStore.getXRayTraceId()` anstelle von Umgebungsvariablen zu

Eine ausführliche Anleitung finden Sie in der Dokumentation zur Node.js Laufzeit von Lambda Managed Instances.

Dateikonflikte (Python-Laufzeit)

Problem: Ihre Python-Funktion liest falsche Daten aus Dateien in`/tmp`.

Ursache: Das **`/tmp`** Verzeichnis wird von mehreren Prozessen gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen.

Lösung:

- Verwenden Sie eindeutige Dateinamen mit Anforderungs-IDs: `/tmp/request_{context.request_id}.txt`
- Verwenden Sie das Sperren von Dateien mit `fcntl.flock()`
- Löschen Sie temporäre Dateien mit `os.remove()` after use

Eine ausführliche Anleitung finden Sie in der Dokumentation zur Python-Laufzeit für Lambda Managed Instances.

Leistungsprobleme

Hohe Speicherauslastung

Problem: Bei Ihren Funktionen kommt es zu einer hohen Speicherauslastung oder es kommt zu Fehlern aufgrund unzureichenden Speichers.

Ursache: Jede gleichzeitige Anforderung in Python wird in einem separaten Prozess mit eigenem Speicherplatz ausgeführt. Die gesamte Speichernutzung entspricht dem Arbeitsspeicher pro Prozess multipliziert mit gleichzeitigen Prozessen.

Lösung:

- Überwachen Sie die Metrik in `MemoryUtilization` CloudWatch
- Reduzieren Sie die `MaxConcurrency` Einstellung, wenn sich die Speicherauslastung dem Speicherlimit der Funktion nähert
- Erhöhen Sie die Speicherzuweisung der Funktion, um eine höhere Parallelität zu unterstützen
- Optimieren Sie die Speichernutzung, indem Sie Daten bei Bedarf statt während der Initialisierung laden

Inkonsistente Leistung

Problem: Die Funktionsleistung variiert erheblich zwischen den Aufrufen.

Ursache: Lambda wählt möglicherweise je nach Verfügbarkeit unterschiedliche Instanztypen aus, oder Funktionen werden möglicherweise auf Instances mit unterschiedlicher Ressourcenverfügbarkeit ausgeführt.

Lösung:

- Zulässige Instanztypen angeben: Wenn Sie bestimmte Leistungsanforderungen haben, konfigurieren Sie zulässige Instance-Typen in Ihrem Kapazitätsanbieter, um die Instance-Typen einzuschränken, die Lambda auswählen kann.
- Überwachen Sie Metriken auf Instanzebene: Verfolgen Sie `CPUUtilization` diese `MemoryUtilization` auf Ebene des Kapazitätsanbieters, um Ressourcenbeschränkungen zu identifizieren.
- Überprüfen Sie die Kapazitätskennzahlen: Prüfen Sie `vCPUAvailable` und stellen `MemoryAvailable` Sie sicher, dass auf Ihren Instances ausreichend Ressourcen verfügbar sind.

Probleme mit dem Kapazitätsanbieter

Die Funktionsversion wird nicht AKTIV

Problem: Ihre Funktionsversion bleibt nach der Veröffentlichung im Status Ausstehend.

Ursache: Lambda startet Managed Instances und startet Ausführungsumgebungen. Dieser Vorgang benötigt Zeit, insbesondere für die erste Funktionsversion auf einem neuen Kapazitätsanbieter.

Lösung:

Warten Sie, bis Lambda den Initialisierungsvorgang abgeschlossen hat. Lambda startet standardmäßig drei Instances aus Gründen der AZ-Resilienz und startet drei Ausführungsumgebungen, bevor Ihre Funktionsversion als AKTIV markiert wird. Dies dauert in der Regel mehrere Minuten.

Der Kapazitätsanbieter kann nicht gelöscht werden

Problem: Sie erhalten eine Fehlermeldung, wenn Sie versuchen, einen Kapazitätsanbieter zu löschen.

Ursache: Sie können einen Kapazitätsanbieter nicht löschen, dem Funktionsversionen zugeordnet sind.

Lösung:

1. Identifizieren Sie alle Funktionsversionen mithilfe des Kapazitätsanbieters mit der `ListFunctionVersionsByCapacityProvider` API.
2. Löschen oder aktualisieren Sie diese Funktionsversionen, um die Zuordnung zum Kapazitätsanbieter zu entfernen.
3. Versuchen Sie erneut, den Kapazitätsanbieter zu löschen.

Allgemeine Fehlermeldungen beim Veröffentlichen von Funktionen

Problem: Beim Veröffentlichen von Funktionen treten generische Fehlermeldungen wie „Beim Publizieren ist ein interner Fehler aufgetreten“ auf.

Lösung:

- Überprüfen Sie die IAM-Berechtigungen: Stellen Sie sicher, dass Sie über die entsprechenden `lambda:PassCapacityProvider` Berechtigungen für den Kapazitätsanbieter verfügen, den Sie verwenden möchten.
- Überprüfen Sie die Konfiguration des Kapazitätsanbieters: Stellen Sie mithilfe der `GetCapacityProvider` API sicher, dass sich Ihr Kapazitätsanbieter im Status ACTIVE befindet.
- VPC-Konfiguration überprüfen: Stellen Sie sicher, dass die in Ihrem Kapazitätsanbieter angegebenen Subnetze und Sicherheitsgruppen korrekt konfiguriert sind und darauf zugegriffen werden kann.
- AWS CloudTrail Protokolle überprüfen: Überprüfen Sie die CloudTrail Protokolle auf detaillierte Fehlerinformationen zum fehlgeschlagenen Vorgang.

Probleme bei der Überwachung und Beobachtbarkeit

Fehlende Metriken CloudWatch

Problem: Sie sehen keine erwarteten Metriken CloudWatch für Ihren Kapazitätsanbieter oder Ihre Funktionen.

Ursache: Metriken werden in Intervallen von 1 Minute veröffentlicht. Bei neuen Kapazitätsanbietern oder Funktionen sind Metriken möglicherweise nicht sofort verfügbar.

Lösung:

Warten Sie nach der Veröffentlichung einer Funktionsversion mindestens 5 bis 10 Minuten, bevor Sie erwarten, dass Metriken angezeigt werden CloudWatch. Vergewissern Sie sich, dass Sie den richtigen Namespace (`AWS/Lambda`) und die richtigen Dimensionen (`CapacityProviderName`, `FunctionName`, oder`InstanceType`) verwenden.

Protokolle können nicht gefunden CloudWatch werden

Problem: Ihre Funktion wird erfolgreich ausgeführt, Sie können jedoch keine Protokolle in CloudWatch den Protokollen finden.

Ursache: Lambda Managed Instances werden in Ihrer VPC ausgeführt und benötigen Netzwerkkonnektivität, um Protokolle an Logs zu CloudWatch senden. Ohne eine ordnungsgemäße VPC-Konnektivitätskonfiguration können Ihre Funktionen den CloudWatch Logs-Service-Endpunkt nicht erreichen.

Lösung:

Konfigurieren Sie die VPC-Konnektivität, damit Ihre Funktionen Protokolle an CloudWatch Logs senden können. Sie haben drei Möglichkeiten:

Option 1: VPC-Endpunkt für CloudWatch Protokolle (für die Produktion empfohlen)

1. Öffnen Sie die Amazon VPC-Konsole unter console.aws.amazon. com/vpc/.
2. Wählen Sie im Navigationsbereich Endpunkte aus.
3. Wählen Sie Endpunkt erstellen aus.
4. Wählen Sie bei Service category (Servicekategorie) die Option AWS services (-Services) aus.
5. Wählen Sie für den Servicenamen die Option `com.amazonaws.region.logs` (durch Ihre AWS Region `region` ersetzen) aus.

6. Wählen Sie für VPC die von Ihrem Kapazitätsanbieter verwendete VPC aus.
7. Wählen Sie für Subnetze die Subnetze aus, in denen Sie Endpunkt-Netzwerkschnittstellen erstellen möchten. Wählen Sie Subnetze in mehreren Availability Zones, um eine hohe Verfügbarkeit zu erzielen.
8. Wählen Sie für Sicherheitsgruppen Sicherheitsgruppen aus, die eingehenden HTTPS-Verkehr (Port 443) aus der Sicherheitsgruppe Ihrer Funktion zulassen.
9. Aktivieren Sie Private DNS für den Endpunkt.
10. Wählen Sie Endpunkt erstellen aus.

Option 2: Öffentliches Subnetz mit Internet-Gateway

Wenn Ihr Kapazitätsanbieter öffentliche Subnetze verwendet, stellen Sie Folgendes sicher:

1. Ein Internet-Gateway ist an Ihre VPC angeschlossen
2. Die Routing-Tabelle leitet `0.0.0.0/0` den Verkehr an das Internet-Gateway weiter
3. Sicherheitsgruppen ermöglichen ausgehenden HTTPS-Verkehr auf Port 443

Option 3: Privates Subnetz mit NAT-Gateway

Wenn Ihr Kapazitätsanbieter private Subnetze verwendet, stellen Sie Folgendes sicher:

1. Ein NAT-Gateway ist in einem öffentlichen Subnetz vorhanden
2. Die Routing-Tabelle des privaten Subnetzes leitet den `0.0.0.0/0` Datenverkehr an das NAT-Gateway weiter
3. Die öffentliche Subnetz-Routing-Tabelle leitet den `0.0.0.0/0` Datenverkehr an ein Internet-Gateway weiter
4. Sicherheitsgruppen ermöglichen ausgehenden HTTPS-Verkehr auf Port 443

Eine ausführliche Anleitung zu VPC-Konnektivitätsoptionen finden Sie unter VPC-Konnektivität für Lambda Managed Instances.

Schwierigkeiten beim Korrelieren von Protokollen aus gleichzeitigen Anfragen

Problem: Die Protokolle verschiedener Anfragen sind verschachtelt, was es schwierig macht, einzelne Anfragen nachzuverfolgen.

Ursache: Das Verschachteln von Protokollen wird erwartet und ist das Standardverhalten in Systemen mit mehreren gleichzeitigen Vorgängen.

Lösung:

- Verwenden Sie die strukturierte Protokollierung im JSON-Format: Schließen Sie die Anforderungs-ID in alle Protokollanweisungen ein
- Java: Verwenden Sie Log4j mit, um die `ThreadContext` Anforderungs-ID automatisch einzubeziehen
- Node.js: **`console.log()`** Mit JSON-Formatierung verwenden und einschließen `InvokeStore.getRequestId()`
- Python: Verwenden Sie das Standard-Logging-Modul mit JSON-Formatierung und schließen Sie Folgendes ein `context.request_id`

Eine ausführliche Anleitung finden Sie auf den laufzeitspezifischen Dokumentationsseiten.

Weitere Hilfe

Wenn Sie nach dem Ausprobieren dieser Lösungen weiterhin Probleme haben:

1. CloudWatch Kennzahlen überprüfen: Überprüfen Sie die Kennzahlen zum Kapazitätsanbieter und zur Ausführungsumgebung, um Ressourcenbeschränkungen oder Skalierungsprobleme zu identifizieren.
2. AWS CloudTrail Protokolle überprüfen: In den CloudTrail Protokollen finden Sie detaillierte Informationen zu API-Aufrufen und Fehlern.
3. Wenden Sie sich an den AWS Support: Wenn Sie das Problem nicht lösen können, kontaktieren Sie den AWS Support mit Einzelheiten zur Konfiguration Ihres Kapazitätsanbieters, zur Funktionskonfiguration und zu den spezifischen Fehlermeldungen, auf die Sie stoßen.

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Machen Sie sich mit der Skalierung für Lambda Managed Instances vertraut
- Lesen Sie die laufzeitspezifischen Anleitungen für Java und Python Node.js ???
- Überwachen Sie von Lambda verwaltete Instances mit Metriken CloudWatch

- Informieren Sie sich über bewährte Methoden für von Lambda verwaltete Instances

Verwaltung der Abhängigkeiten von Lambda Event Function mit Ebenen

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

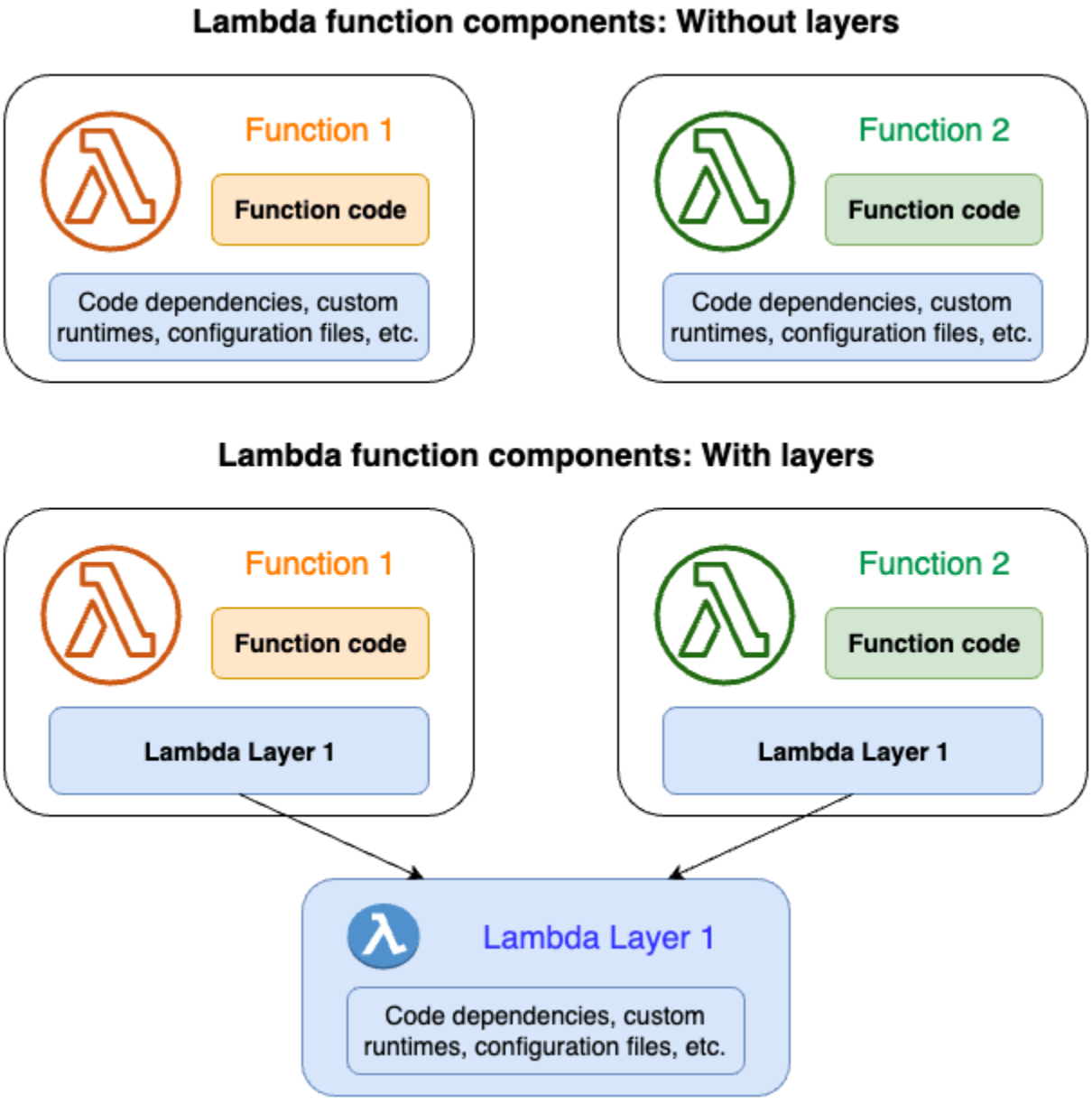
Die Verwendung von Ebenen ermöglicht Folgendes:

- Verringern der Größe Ihrer Bereitstellungspakete: Anstatt all Ihre Event-Funktions-Abhängigkeiten zusammen mit Ihrem Event-Function-Code in Ihr Bereitstellungspaket aufzunehmen, fügen Sie sie in eine Ebene ein. So bleiben Bereitstellungspakete klein und übersichtlich.
- Trennen von grundlegender Funktionslogik und Abhängigkeiten: Mithilfe von Ebenen können Sie Ihre Event-Funktions-Abhängigkeiten unabhängig von Ihrem Event-Funktionscode aktualisieren und umgekehrt. Dies fördert die Trennung der einzelnen Bereiche und hilft Ihnen, sich auf Ihre Event-Function-Logik zu konzentrieren.
- Verwenden von Abhängigkeiten für mehrere Funktionen: Nachdem Sie eine Ebene erstellt haben, können Sie sie auf eine beliebige Anzahl von Event-Funktionen in Ihrem Konto anwenden. Ohne Ebenen müssen die gleichen Abhängigkeiten in jedes einzelne Bereitstellungspaket eingefügt werden.
- Verwenden des Code-Editors der Lambda-Konsole: Der Code-Editor ist ein nützliches Tool, um kleinere Code-Updates für Event-Funktionen schnell zu testen. Sie können den Editor jedoch nicht verwenden, wenn Ihr Bereitstellungspaket zu groß ist. Die Verwendung von Ebenen verringert die Paketgröße und kann die Nutzung des Code-Editors ermöglichen.
- Um eine eingebettete SDK-Version zu sperren. Die eingebetteten SDKs können sich ohne vorherige Ankündigung ändern, wenn neue Dienste und Funktionen AWS veröffentlicht werden. Sie können eine Version des SDK sperren, indem Sie eine Lambda-Ebene mit der jeweils benötigten Version erstellen. Die Event-Funktion verwendet dann immer die Version in der Ebene, auch wenn sich die im Service eingebettete Version ändert.

Wenn Sie mit Lambda Event Functions in Go oder Rust arbeiten, empfehlen wir, keine Ebenen zu verwenden. Für Go- und Rust-Ereignisfunktionen stellen Sie Ihren Event-Function-Code als ausführbare Datei bereit, die Ihren kompilierten Event-Funktionscode zusammen mit all seinen

Abhängigkeiten enthält. Wenn Sie Ihre Abhängigkeiten in einer Ebene platzieren, muss Ihre Event-Funktion während der Initialisierungsphase zusätzliche Assemblys manuell laden, was die Kaltstartzeiten verlängern kann. Um eine optimale Leistung für Go und Rust Event Functions zu erzielen, sollten Sie Ihre Abhängigkeiten zusammen mit Ihrem Bereitstellungspaket angeben.

Das folgende Diagramm veranschaulicht die grundlegenden architektonischen Unterschiede zwischen zwei Event Functions, die Abhängigkeiten gemeinsam haben. Bei einer Variante werden Lambda-Ebenen verwendet, bei der anderen nicht.



Wenn Sie einer Event-Funktion eine Ebene hinzufügen, extrahiert Lambda den Layer-Inhalt in das `/opt` Verzeichnis in der Ausführungsumgebung Ihrer Event-Funktion. Alle nativ unterstützten Lambda-Laufzeiten enthalten Pfade zu spezifischen Verzeichnissen innerhalb des Verzeichnisses `/opt`. Dadurch erhält Ihre Event-Funktion Zugriff auf Ihre Layer-Inhalte. Weitere Informationen zu diesen spezifischen Pfaden und zum richtigen Verpacken Ihrer Ebenen finden Sie unter the section called “Verpacken von Ebenen”.

Sie können bis zu fünf Ebenen pro Event-Funktion hinzufügen. Außerdem können Sie Ebenen nur mit Lambda Event Functions verwenden, die als ZIP-Dateiarchiv bereitgestellt werden. Für Event Functions, die als Container-Image definiert sind, packen Sie Ihre bevorzugte Laufzeit und alle Codeabhängigkeiten, wenn Sie das Container-Image erstellen. Weitere Informationen finden Sie im AWS Compute-Blog unter Arbeiten mit Lambda-Layern und Erweiterungen in Container-Images.

Themen

- Verwenden von Ebenen
- Ebenen und Ebenenversionen
- Verpacken Ihres Ebeneninhalts
- Erstellen und Löschen von Ebenen in Lambda
- Hinzufügen von Ebenen zu Event-Funktionen
- Verwenden AWS CloudFormation mit Schichten
- Verwenden AWS SAM mit Schichten

Verwenden von Ebenen

Zum Erstellen einer Ebene müssen Sie Ihre Abhängigkeiten in einer ZIP-Datei verpacken – ähnlich wie beim Erstellen eines normalen Bereitstellungspakets. Genauer gesagt umfasst der allgemeine Prozess der Erstellung und Verwendung von Ebenen die drei folgenden Schritte:

- Verpacken Sie zuerst Ihren Ebeneninhalt. Erstellen Sie also ein ZIP-Dateiarchiv. Weitere Informationen finden Sie unter the section called “Verpacken von Ebenen”.
- Erstellen Sie als Nächstes die Ebene in Lambda. Weitere Informationen finden Sie unter the section called “Erstellen und Löschen von Ebenen”.
- Fügen Sie die Ebene zu Ihren Event-Funktion (en) hinzu. Weitere Informationen finden Sie unter the section called “Hinzufügen von Ebenen”.

Ebenen und Ebenenversionen

Eine Ebenenversion ist eine unveränderliche Momentaufnahme einer spezifischen Version einer Ebene. Wenn Sie eine neue Ebene erstellen, erstellt Lambda eine neue Ebenenversion mit der Versionsnummer 1. Bei jeder Veröffentlichung einer Aktualisierung für die Ebene erhöht Lambda die Versionsnummer und erstellt eine neue Ebenenversion.

Jede Ebenenversion wird durch einen eindeutigen Amazon-Ressourcennamen (ARN) identifiziert. Wenn Sie der Event-Funktion eine Ebene hinzufügen, müssen Sie genau die Layer-Version angeben, die Sie verwenden möchten (z. B.`arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`).

Verpacken Ihres Ebeneninhalts

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

In diesem Abschnitt erfahren Sie, wie Sie Ihren Ebeneninhalt ordnungsgemäß verpacken. Weitere konzeptionelle Informationen zu Ebenen sowie dazu, warum Sie ggf. welche verwenden sollten, finden Sie unter the section called “Layer”.

Der erste Schritt beim Erstellen einer Ebene besteht darin, den gesamten Ebeneninhalt in einem ZIP-Dateiarchiv zu bündeln. Da Lambda-Funktionen unter Amazon Linux ausgeführt werden, muss Ihr Ebeneninhalt in einer Linux-Umgebung kompiliert und erstellt werden können.

Um sicherzustellen, dass Ihr Ebeneninhalt ordnungsgemäß in einer Linux-Umgebung funktioniert, empfiehlt es sich, den Ebeneninhalt mit einem Tool wie Docker zu erstellen.

Themen

- Ebenenpfade für jede Lambda-Laufzeit

Ebenenpfade für jede Lambda-Laufzeit

Wenn Sie einer Funktion eine Ebene hinzufügen, lädt Lambda den Ebeneninhalt in das Verzeichnis `/opt` der Ausführungsumgebung. Für jede Lambda-Laufzeit enthält die Variable `PATH` bereits spezifische Ordnerpfade innerhalb des Verzeichnisses `/opt`. Um sicherzustellen, dass Lambda den Inhalt Ihrer Ebene aufnimmt, müssen die Abhängigkeiten Ihrer ZIP-Datei in einem der folgenden Ordnerpfade enthalten sein:

Laufzeit	Pfad
Node.js	`nodejs/node_modules`
	`nodejs/node18/node_modules` (`NODE_PATH`)
	`nodejs/node20/node_modules` (`NODE_PATH`)
	`nodejs/node22/node_modules` (`NODE_PATH`)
Python	`python`

Laufzeit	Pfad
	`python/lib/ python3.x /site-packages` (Site-Verzeichnisse)
Java	`java/lib` (`CLASSPATH`)
Ruby	`ruby/gems/3.4.0` (`GEM_PATH`)
	`ruby/lib` (`RUBYLIB`)
Alle Laufzeiten	`bin` (`PATH`)
	`lib` (`LD_LIBRARY_PATH`)

Die folgenden Beispiele zeigen, wie Sie die Ordner im ZIP-Archiv Ihrer Ebene strukturieren können.

Node.js

Example Dateistruktur für AWS X-Ray SDK für Node.js

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

Ruby

Example Dateistruktur für das JSON-Gem

```
json.zip
# ruby/gems/3.4.0/
```

```
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

Java

Example Dateistruktur für Jackson-JAR-Datei

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example Dateistruktur für jq-Bibliothek

```
jq.zip
# bin/jq
```

Eine sprachspezifische Anleitung zum Verpacken, Erstellen und Hinzufügen einer Ebene finden Sie auf den folgenden Seiten:

- Node.js – the section called “Layer”
- Python – the section called “Layer”
- Ruby – the section called “Layer”
- Java – the section called “Layer”

Wir raten davon ab, Ebenen zur Verwaltung von Abhängigkeiten für Lambda-Ereignisfunktionen zu verwenden, die in Go und Rust geschrieben wurden. Dies liegt daran, dass in diesen Sprachen geschriebene Lambda-Ereignisfunktionen zu einer einzigen ausführbaren Datei kompiliert werden,

die Sie Lambda bei der Bereitstellung Ihrer Ereignisfunktion zur Verfügung stellen. Diese ausführbare Datei enthält Ihren kompilierten Event-Function-Code zusammen mit all seinen Abhängigkeiten. Die Verwendung von Ebenen verkompliziert nicht nur diesen Vorgang, sondern führt auch zu längeren Kaltstartzeiten, da Ihre Event-Funktionen während der Init-Phase zusätzliche Assemblys manuell in den Speicher laden müssen.

Um externe Abhängigkeiten mit Go und Rust Lambda Event Functions zu verwenden, nehmen Sie sie direkt in Ihr Bereitstellungspaket auf.

Erstellen und Löschen von Ebenen in Lambda

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

In diesem Abschnitt erfahren Sie, wie Sie Ebenen in Lambda erstellen und löschen. Weitere konzeptionelle Informationen zu Ebenen sowie dazu, warum Sie ggf. welche verwenden sollten, finden Sie unter the section called “Layer”.

Nachdem Sie Ihren Layer-Inhalt gepackt haben, besteht der nächste Schritt darin, den Layer in Lambda zu erstellen. In diesem Abschnitt wird nur gezeigt, wie Ebenen mithilfe der Lambda-Konsole oder mithilfe der Lambda-API erstellt und gelöscht werden. Eine Anleitung zum Erstellen einer Ebene mithilfe von AWS CloudFormation finden Sie unter the section called “Schichten mit CloudFormation”. Eine Anleitung zum Erstellen einer Ebene mithilfe von AWS Serverless Application Model (AWS SAM) finden Sie unter the section called “Schichten mit AWS SAM”.

Themen

- Erstellen einer Ebene
- Löschen einer Ebenen-Version

Erstellen einer Ebene

Zum Erstellen einer Ebene können Sie das ZIP-Archiv entweder über Ihren lokalen Computer oder über Amazon Simple Storage Service (Amazon S3) hochladen. Beim Hochladen von Amazon S3 können Sie wählen, ob Lambda den Code in den Lambda-managed Speicher kopiert oder direkt aus Ihrem S3-Bucket darauf verweist. Weitere Informationen finden Sie unter the section called “Self-managed S3-Code-Speicher”.

Lambda extrahiert den Layer-Inhalt in das `/opt` Verzeichnis, wenn die Ausführungsumgebung für die Lambda-Ereignisfunktion eingerichtet wird.

Ebenen können eine oder mehrere Ebenenversionen haben. Wenn Sie eine Ebene erstellen, legt Lambda die Ebenen-Version auf Version 1 fest. Die Berechtigungen für eine bereits vorhandene Ebenenversion können jederzeit geändert werden. Um den Code zu aktualisieren oder andere Konfigurationsänderungen vorzunehmen, müssen Sie allerdings eine neue Version der Ebene erstellen.

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Geben Sie unter Ebenen-Konfiguration unter Name einen Namen für Ihre Ebene ein.
4. (Optional) Geben Sie im Feld Description (Beschreibung) eine Beschreibung für Ihre Ebene ein.
5. Um Ihren Ebenen-Code upzuloaden, führen Sie einen der folgenden Schritte aus:
   - Um eine ZIP-Datei von Ihrem Computer upzuloaden, wählen Sie Eine .zip-Datei hochladen. Wählen Sie Upload, um Ihre lokale ZIP-Datei auszuwählen.
   - Um eine Datei aus Amazon S3 upzuloaden, wählen Sie Hochladen einer Datei aus Amazon S3 aus. Geben Sie dann für Amazon-S3-Link-URL einen Link zu der Datei ein.
6. Wenn Sie sich für den Upload von Amazon S3 entschieden haben, wählen Sie im Codespeichermodus eine der folgenden Optionen:
   - Kopiermodus (Standard)
   - Referenzmodus
7. (Optional) Für Kompatible Architekturen, wählen Sie einen Wert oder beide Werte aus. Weitere Informationen finden Sie unter the section called “Befehlssätze (ARM/x86)”.
8. (Optional) Wählen Sie unter Kompatible Laufzeiten die Laufzeiten aus, mit denen Ihre Ebene kompatibel ist.
9. (Optional) Für License (Lizenz) geben Sie alle erforderlichen Lizenzinformationen ein.
10. Wählen Sie Erstellen aus.

Alternativ können Sie den Befehl publish-layer-version AWS Command Line Interface (CLI) ausführen. Geben Sie beim Hochladen von Amazon S3 `S3ObjectStorageMode=REFERENCE` an, ob Sie den selbstverwalteten S3-Codespeicher verwenden möchten, oder lassen Sie ihn weg (StandardeinstellungCOPY), um Speicher zu verwenden. Lambda-managed Beispiel:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip --
compatible-runtimes python3.14
```

Bei jeder Ausführung von `publish-layer-version` erstellt Lambda eine neue Version der Ebene.

Löschen einer Ebenen-Version

Verwenden Sie den API-Vorgang, um eine Layer-Version zu löschen. DeleteLayerVersion Führen Sie beispielsweise den AWS CLI Befehl delete-layer-version mit dem angegebenen Layer-Namen und der Layer-Version aus.

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

Wenn Sie eine Layer-Version löschen, können Sie keine neuen Ereignisfunktionen mehr für deren Verwendung konfigurieren. Jede Ereignisfunktion, die bereits auf die gelöschte Layer-Version verweist, wird jedoch weiterhin mit diesem Layer-Inhalt ausgeführt. Lambda verwendet niemals Versionsnummern für einen Layer-Namen wieder.

Auswirkungen des Löschens von Ebenen auf das Speicherkontingent

Wie sich das Löschen einer Layer-Version auf das Speicherkontingent auswirkt, hängt vom Speichermodus ab, der von der Ebene verwendet wird, und von den Funktionen, die darauf verweisen.

Die Ebene verwendet Lambda-managed Speicher (COPY)

> Wenn Sie eine Layer-Version löschen, die Lambda-managed Speicherplatz verwendet, wird der Layer-Code aus dem Lambda-managed Speicherkontingent Ihres Kontos entfernt. Jede Event-Funktion, die immer noch auf die gelöschte Layer-Version verweist, hat jedoch weiterhin Zugriff auf den Layer-Inhalt. Der Layer-Inhalt wird auf das Größenkontingent für das Bereitstellungspaket (250 MB entpackt) der Event-Funktion angerechnet, aber nicht mehr auf das Speicherkontingent auf Lambda-managed Kontoebene.

Layer verwendet selbstverwalteten S3-Speicher (REFERENZ)

> Ebenen, die selbstverwalteten S3-Codespeicher verwenden, verbrauchen keinen Lambda-managed Speicherplatz, sodass das Löschen dieser Ebenen keine Auswirkungen auf Ihr Lambda-managed Speicherkontingent hat. Event-Funktionen, die auf eine gelöschte selbstverwaltete Layer-Version verweisen, werden weiterhin normal ausgeführt. Wenn Sie jedoch auch das Quellobjekt aus Ihrem S3-Bucket löschen oder Lambda den Zugriff darauf entziehen, können Sie die Konfiguration der Ereignisfunktion nicht mehr mit dieser Ebene aktualisieren.

> **Note**
>
> Durch das Löschen einer Layer-Version werden niemals Event-Funktionen deaktiviert, die darauf verweisen. Event-Funktionen bleiben erhalten, `Active` unabhängig davon, ob die Ebene S3-Speicher verwendet Lambda-managed oder selbstverwalteten S3-Speicher verwendet hat. Der `Inactive` Status gilt nur, wenn Lambda den Zugriff auf den eigenen Quellcode einer Event-Funktion verliert, der in einem selbstverwalteten S3-Bucket gespeichert ist.

Hinzufügen von Ebenen zu Event-Funktionen

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

In diesem Abschnitt wird erklärt, wie Sie einer Lambda-Ereignisfunktion eine Ebene hinzufügen. Weitere konzeptionelle Informationen zu Ebenen sowie dazu, warum Sie ggf. welche verwenden sollten, finden Sie unter the section called “Layer”.

Bevor Sie eine Ereignisfunktion für die Verwendung einer Ebene konfigurieren können, müssen Sie:

- Verpacken Ihres Ebeneninhalts
- Erstellen einer Ebene in Lambda
- Stellen Sie sicher, dass Sie berechtigt sind, die GetLayerVersionAPI für die Layer-Version aufzurufen. Für Event Functions in Ihrem AWS-Konto müssen Sie diese Berechtigung in Ihrer Benutzerrichtlinie haben. Um eine Ebene in einem anderen Konto zu verwenden, muss der Eigentümer des anderen Kontos Ihrem Konto die Berechtigung in einer ressourcenbasierten Richtlinie erteilen. Beispiele finden Sie unter the section called “Ebenenzugriff für andere Konten”.

Sie können einer Event-Funktion bis zu fünf Ebenen hinzufügen. Die entpackte Gesamtgröße der Event Function und aller Ebenen darf das Größenkontingent für das entpackte Bereitstellungspaket von 250 MB nicht überschreiten. Weitere Informationen finden Sie unter Lambda-Kontingente.

Ihre Event-Funktionen können weiterhin jede Layer-Version verwenden, die Sie bereits hinzugefügt haben, auch nachdem diese Layer-Version gelöscht wurde oder nachdem Ihre Zugriffsberechtigung für den Layer widerrufen wurde. Sie können jedoch keine neue Event-Funktion erstellen, die eine gelöschte Layer-Version verwendet.

Um einer Event-Funktion eine Ebene hinzuzufügen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Event-Funktion.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen eine Ebenenquelle aus:

    a. AWS Ebenen: Wählen Sie aus der Liste der AWS verwalteten Erweiterungen aus.

   b. Benutzerdefinierte Ebenen: Wählen Sie eine Ebene aus, die in Ihrem AWS-Konto erstellt wurde.
   c. Geben Sie einen ARN an: Um einen Layer aus einem anderen AWS-Konto, z. B. einer Drittanbieter-Erweiterung, zu verwenden, geben Sie den Amazon-Ressourcennamen (ARN) ein.
5. Wählen Sie Hinzufügen aus.

Die Reihenfolge, in der Sie die Ebenen hinzufügen, ist die Reihenfolge, in der Lambda den Ebeneninhalt in der Ausführungsumgebung zusammenführt. Sie können die Reihenfolge für Zusammenführungen der Layers mit der -Konsole ändern.

Um die Reihenfolge der Layer-Zusammenführung für Ihre Event-Funktion (Konsole) zu aktualisieren

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die zu konfigurierende Event-Funktion aus.
3. Wählen Sie unter Ebenen die Option Bearbeiten
4. Wählen Sie eine der Ebenen aus.
5. Klicken Sie aufFrüher MergeoderSpäter Zusammenführungenum die Reihenfolge der Ebenen anzupassen.
6. Wählen Sie Speichern.

Ebenen werden versioniert. Der Inhalt jeder Ebenenversion ist unveränderlich. Der Besitzer einer Ebene kann neue Ebenenversionen veröffentlichen, um aktualisierte Inhalte bereitzustellen. Sie können die Konsole verwenden, um die Layer-Version zu aktualisieren, die Ihren Event-Funktionen zugeordnet ist.

Um Layer-Versionen für Ihre Event-Funktion (Konsole) zu aktualisieren

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie die Ebene aus, für die Sie die Version aktualisieren möchten.
3. Wählen Sie die Registerkarte Funktionen mit dieser Version aus.
4. Wählen Sie die Event-Funktionen aus, die Sie ändern möchten, und klicken Sie dann auf Bearbeiten.
5. Wählen Sie unter Ebenenversion die Ebenenversion aus, zu der Sie wechseln möchten.
6. Klicken Sie auf Funktionen aktualisieren.

Sie können die Layer-Versionen der Event-Funktion nicht AWS kontenübergreifend aktualisieren.

Suche nach Ebeneninformationen

Verwenden Sie die ListLayersAPI, um Ebenen in Ihrem Konto zu finden, die mit der Laufzeit Ihrer Event Function kompatibel sind. Sie können beispielsweise den folgenden Befehl list-layers AWS Command Line Interface (CLI) verwenden:

```
aws lambda list-layers --compatible-runtime python3.14
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2025-04-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.14"
                ]
            }
        }
    ]
}
```

Wenn Sie alle Ebenen in Ihrem Konto auflisten möchten, lassen Sie die Option `--compatible-runtime` weg. Die Antwortdetails enthalten jeweils die neueste Version der einzelnen Ebenen.

Sie können die neueste Version eines Layers auch mithilfe der ListLayerVersionsAPI abrufen. Sie können z. B. den folgenden CLI-Befehl vom Typ `list-layer-versions` verwenden:

```
aws lambda list-layer-versions --layer-name my-layer
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

Verwenden AWS CloudFormation mit Schichten

Sie können verwenden CloudFormation , um eine Ebene zu erstellen und die Ebene mit Ihrer Lambda-Ereignisfunktion zu verknüpfen. Die folgende Beispielvorlage erstellt eine Ebene mit dem Namen **`my-lambda-layer`** und ordnet die Ebene mithilfe der Eigenschaft Layers der Event-Funktion zu.

In diesem Beispiel gibt die Vorlage den Amazon-Ressourcennamen (ARN) einer bestehenden IAM-Ausführungsrolle an. Sie können mit der Ressource CloudFormation AWS: :IAM: :Role auch eine neue Ausführungsrolle in der Vorlage erstellen.

Ihre Event-Funktion benötigt keine speziellen Berechtigungen, um Ebenen zu verwenden.

```
---
Description: CloudFormation Template for Lambda Event Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: amzn-s3-demo-bucket
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Role: arn:aws:iam::111122223333:role/my_lambda_role
      Layers:
        - !Ref MyLambdaLayer
```

Sie können sie auch AWS Serverless Application Model zur Verwaltung von Ebenen verwenden. Weitere Informationen finden Sie unter the section called “Schichten mit AWS SAM”.

Verwenden AWS SAM mit Schichten

Sie können das AWS Serverless Application Model (AWS SAM) verwenden, um die Erstellung von Ebenen in Ihrer Anwendung zu automatisieren. Der `AWS::Serverless::LayerVersion` Ressourcentyp erstellt eine Layer-Version, auf die Sie in Ihrer Lambda Event Function-Konfiguration verweisen können.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Event Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://amzn-s3-demo-bucket/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://amzn-s3-demo-bucket/my-function
      Layers:
        - !Ref MyLambdaLayer
```

Erweitern Sie Lambda-Ereignisfunktionen mithilfe von Lambda-Erweiterungen

Sie können Lambda-Erweiterungen verwenden, um Ihre Lambda-Ereignisfunktionen zu erweitern. Verwenden Sie beispielsweise Lambda-Erweiterungen, um Event Functions in Ihre bevorzugten Tools für Überwachung, Observability, Sicherheit und Governance zu integrieren. Sie können aus

einer Vielzahl von Tools wählen, die von AWS Lambda -Partnern bereitgestellt werden, oder Sie können Ihre eigenen Lambda-Erweiterungen erstellen.

Lambda unterstützt externe und interne Erweiterungen. Eine externe Erweiterung wird als unabhängiger Prozess in der Ausführungsumgebung ausgeführt und läuft weiter, nachdem der Aufruf der Event Function vollständig verarbeitet wurde. Da Erweiterungen als separate Prozesse ausgeführt werden, können Sie sie in einer anderen Sprache als die Event-Funktion schreiben. Alle Laufzeiten der Lambda-Event-Funktion unterstützen Erweiterungen.

Eine interne Erweiterung wird als Teil des Laufzeitprozesses ausgeführt. Ihre Event-Funktion greift mithilfe von Wrapper-Skripten oder prozessinternen Mechanismen wie auf interne Erweiterungen zu. `JAVA_TOOL_OPTIONS` Weitere Informationen finden Sie unter Ändern der Laufzeitumgebung.

Sie können Erweiterungen zu einer Event-Funktion hinzufügen, indem Sie die Lambda-Konsole, die Dienste AWS Command Line Interface (AWS CLI) oder Infrastructure as Code (IaC) und Tools wie CloudFormation, AWS Serverless Application Model (AWS SAM) und Terraform verwenden.

Ihnen wird die Ausführungszeit berechnet, die von der Erweiterung verbraucht wird (in Schritten von 1 ms). Es fallen keine Kosten für die Installation eigener Erweiterungen an. Weitere Informationen zur Preisgestaltung für Erweiterungen finden Sie unter AWS Lambda -Preise. Preisinformationen zu Partnererweiterungen finden Sie auf den Websites dieser Partner. Eine Liste der offiziellen Partnererweiterungen finden Sie unter the section called “Partner für Erweiterungen”.

Themen

- Ausführungsumgebung
- Auswirkungen auf Leistung und Ressourcen
- Berechtigungen
- Konfigurieren von Lambda-Erweiterungen
- AWS Lambda Partner für Erweiterungen
- Verwendung der Lambda-Erweiterungs-API zur Erstellung von Erweiterungen
- Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API

Ausführungsumgebung

Lambda ruft Ihre Event Function in einer Ausführungsumgebung auf, die eine sichere und isolierte Laufzeitumgebung bietet. Die Ausführungsumgebung verwaltet die Ressourcen, die für die

Ausführung Ihrer Event-Funktion erforderlich sind, und bietet Lebenszyklusunterstützung für die Laufzeit und die Erweiterungen der Event-Funktion.

Der Lebenszyklus der Ausführungsumgebung umfasst die folgenden Phasen:

- **`Init`**: In dieser Phase erstellt oder hebt Lambda eine Ausführungsumgebung mit den konfigurierten Ressourcen auf, lädt den Code für die Funktion und alle Ebenen herunter, initialisiert alle Erweiterungen, initialisiert die Laufzeit und führt dann den Initialisierungscode der Funktion (der Code außerhalb des Haupthandlers) aus. Die Phase `Init` erfolgt entweder während des ersten Aufrufs oder vor Funktionsaufrufen, wenn Sie die bereitgestellte Parallelität aktiviert haben.

  Die `Init`-Phase ist in drei Unterphasen unterteilt: `Extension init`, `Runtime init` und `Function init`. Diese Unterphasen stellen sicher, dass alle Erweiterungen und die Laufzeit ihre Einrichtungs-Aufgaben abschließen, bevor der Funktionscode ausgeführt wird.

  Wenn SnapStart aktiviert ist, findet die `Init`-Phase statt, wenn Sie eine Funktionsversion veröffentlichen. Lambda speichert einen Snapshot des Arbeitsspeichers und des Festplattenzustands der initialisierten Ausführungsumgebung, speichert den verschlüsselten Snapshot und speichert ihn im Cache für den Zugriff mit geringer Latenz. Wenn Sie einen Laufzeit-Hook vor dem Prüfpunkt haben, wird der Code am Ende der `Init`-Phase ausgeführt.
- **`Restore`**(SnapStart nur): Wenn Sie eine SnapStartFunktion zum ersten Mal aufrufen und die Funktion skaliert wird, nimmt Lambda neue Ausführungsumgebungen aus dem persistenten Snapshot wieder auf, anstatt die Funktion von Grund auf neu zu initialisieren. Wenn Sie über einen Laufzeit-Hook nach der Wiederherstellung verfügen, wird der Code am Ende der `Restore`-Phase ausgeführt. Die Dauer von Laufzeit-Hooks nach der Wiederherstellung wird Ihnen in Rechnung gestellt. Die Laufzeit muss geladen werden und Laufzeit-Hooks nach der Wiederherstellung müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine. SnapStartTimeoutException Wenn die `Restore`-Phase abgeschlossen ist, ruft Lambda den Funktionshandler (Invoke-Phase) auf.
- **`Invoke`**: In dieser Phase ruft Lambda den Funktionshandler auf. Nachdem die Funktion vollständig ausgeführt wurde, bereitet sich Lambda auf die Verarbeitung eines weiteren Funktionsaufrufs vor.
- **`Shutdown`**: Diese Phase wird ausgelöst, wenn die Lambda-Funktion für einen bestimmten Zeitraum keine Aufrufe empfängt. In dieser `Shutdown`-Phase fährt Lambda die Laufzeit herunter, warnt die Erweiterungen, damit sie sauber beendet werden können und entfernt dann die Umgebung. Lambda sendet ein `Shutdown`-Ereignis an jede Erweiterung, das der Erweiterung mitteilt, dass die Umgebung beendet wird.

Während der `Init`-Phase extrahiert Lambda Ebenen, die Erweiterungen enthalten, in das `/opt`-Verzeichnis in der Ausführungsumgebung. Lambda sucht im `/opt/extensions/`-Verzeichnis nach Erweiterungen, interpretiert jede Datei als ausführbaren Bootstrap zum Starten der Erweiterung und startet alle Erweiterungen parallel.

Auswirkungen auf Leistung und Ressourcen

Die Größe der Erweiterungen Ihrer Event-Funktion wird auf die Größenbeschränkung für das Bereitstellungspaket angerechnet. Bei einem ZIP-Dateiarchiv darf die entpackte Gesamtgröße der Event Function und aller Erweiterungen die Größenbeschränkung für das entpackte Bereitstellungspaket von 250 MB nicht überschreiten.

Erweiterungen können sich auf die Leistung Ihrer Event-Funktion auswirken, da sie Event-Funktions-Ressourcen wie CPU, Arbeitsspeicher und Speicher gemeinsam nutzen. Wenn eine Erweiterung beispielsweise rechenintensive Operationen ausführt, kann es sein, dass sich die Ausführungsdauer Ihrer Event-Funktion verlängert.

Jede Erweiterung muss ihre Initialisierung abschließen, bevor Lambda die Event-Funktion aufruft. Daher kann eine Erweiterung, die viel Initialisierungszeit in Anspruch nimmt, die Latenz des Aufrufs der Ereignisfunktion erhöhen.

Um die zusätzliche Zeit zu messen, die die Erweiterung nach der Ausführung der Ereignisfunktion benötigt, können Sie die `PostRuntimeExtensionsDuration` Funktionsmetrik verwenden. Um die Zunahme des verwendeten Speichers zu messen, können Sie die `MaxMemoryUsed`-Metrik verwenden. Um die Auswirkungen einer bestimmten Erweiterung zu verstehen, können Sie verschiedene Versionen Ihrer Event-Funktionen nebeneinander ausführen.

> **Note**
>
> `MaxMemoryUsed` ist eine der Metriken, die von Lambda Insights erfasst werden, und keine native Lambda-Metrik.

Berechtigungen

Erweiterungen haben Zugriff auf dieselben Ressourcen wie Event Functions. Da Erweiterungen in derselben Umgebung wie die Ereignisfunktion ausgeführt werden, werden die Berechtigungen zwischen der Ereignisfunktion und der Erweiterung gemeinsam genutzt.

Für ein ZIP-Dateiarchiv können Sie eine CloudFormation Vorlage erstellen, um das Anhängen derselben Erweiterungskonfiguration — einschließlich AWS Identity and Access Management (IAM) - Berechtigungen — an mehrere Ereignisfunktionen zu vereinfachen.

Konfigurieren von Lambda-Erweiterungen

Konfigurieren von Erweiterungen (ZIP-Dateiarchiv)

Sie können Ihrer Lambda Event Function eine Erweiterung als Lambda-Schicht hinzufügen. Mithilfe von Ebenen können Sie Erweiterungen unternehmensweit oder mit der gesamten Lambda-Entwickler-Community teilen. Sie können einer Ebene eine oder mehrere Erweiterungen hinzufügen. Sie können bis zu 10 Erweiterungen für eine Event-Funktion registrieren.

Sie fügen die Erweiterung mit derselben Methode zu Ihrer Event-Funktion hinzu, wie Sie es für jede Ebene tun würden. Weitere Informationen finden Sie unter the section called “Layer”.

Fügen Sie Ihrer Event-Funktion (Konsole) eine Erweiterung hinzu

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Event-Funktion aus.
3. Wählen Sie die Registerkarte Code aus, falls sie noch nicht ausgewählt ist.
4. Wählen Sie unter Ebenen die Option Bearbeiten aus.
5. Geben Sie unter Choose a layer (Ebene auswählen) Specify an ARN (Einen ARN angeben).
6. Geben Sie unter Specify an ARN (Einen ARN angeben), den Amazon-Ressourcennamen (ARN) einer Erweiterungsebene ein.
7. Wählen Sie Add aus.

Verwenden von Erweiterungen in Container-Images

Sie können Ihrem Container-Image Erweiterungen hinzufügen. Die ENTRYPOINT-Container-Image-Einstellung legt den Hauptprozess für die Event-Funktion fest. Konfigurieren Sie die ENTRYPOINT-Einstellung im Dockerfile oder als Override in der Konfiguration der Event-Funktion.

Sie können mehrere Prozesse in einem Container ausführen. Lambda verwaltet den Lebenszyklus des Hauptprozesses und aller zusätzlichen Prozesse. Lambda verwendet die Erweiterungs-API, um den Erweiterungslebenszyklus zu verwalten.

Beispiel: Hinzufügen einer externen Erweiterung

Eine externe Erweiterung wird in einem separaten Prozess als die Lambda-Ereignisfunktion ausgeführt. Lambda startet einen Prozess für jede Erweiterung im `/opt/extensions/`-Verzeichnis. Lambda verwendet die Erweiterungs-API, um den Erweiterungslebenszyklus zu verwalten. Nachdem

die Event-Funktion vollständig ausgeführt wurde, sendet Lambda ein `Shutdown` Ereignis an jede externe Erweiterung.

Example Hinzufügen einer externen Erweiterung zu einem Python-Basis-Image

```
FROM public.ecr.aws/lambda/python:3.11

# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt/extensions
ADD my-extension.zip /opt/extensions
CMD python ./my-function.py
```

Nächste Schritte

Um mehr über Erweiterungen zu erfahren, empfehlen wir die folgenden Ressourcen:

- Ein grundlegendes funktionierendes Beispiel finden Sie unter Building Extensions for AWS Lambda im AWS Compute-Blog.
- Informationen zu Erweiterungen, die AWS Lambda Partners anbietet, finden Sie im AWS Compute-Blog unter Introducing AWS Lambda Extensions.
- Verfügbare Beispielerweiterungen und Wrapper-Skripte finden Sie unter AWS Lambda Erweiterungen im AWS GitHub Samples-Repository.

AWS Lambda Partner für Erweiterungen

AWS Lambda hat sich mit mehreren Drittanbietern zusammengetan, um Erweiterungen für die Integration in Ihre Lambda Event Functions bereitzustellen. In der folgenden Liste werden Erweiterungen von Drittanbietern aufgeführt, die Sie jederzeit verwenden können.

- AppDynamics— Bietet automatische Instrumentierung von Node.js oder Python-Lambda-Ereignisfunktionen und bietet so Transparenz und Warnmeldungen zur Leistung von Eventfunktionen.
- Axiom— Stellt Dashboards zur Überwachung der Leistung von Lambda Event Function und zur Zusammenfassung von Metriken auf Systemebene bereit.
- Datadog – Bietet umfassende Echtzeit-Transparenz für Ihre Serverless-Anwendungen durch die Verwendung von Metriken, Nachverfolgungen und Protokollen.
- Dynatrace— Bietet Einblick in Traces und Metriken und nutzt KI für die automatisierte Fehlererkennung und Ursachenanalyse im gesamten Anwendungsstapel.
- Elastic – Bietet Application Performance Monitoring (APM) zur Identifizierung und Behebung von Ursachenproblemen mithilfe korrelierter Traces, Metriken und Logs.
- Epsagon— Hört auf Aufrufereignisse, speichert Traces und sendet sie parallel zu den Ausführungen der Lambda-Ereignisfunktion.
- Fastly— Schützt Ihre Lambda Event Functions vor verdächtigen Aktivitäten wie Injection-Angriffen, Kontoübernahmen durch Credential-Stuffing, böswilligen Bots und API-Missbrauch.
- HashiCorp Vault— Verwaltet geheime Daten und stellt sie Entwicklern zur Verfügung, damit sie innerhalb des Event-Funktionscodes verwendet werden können, ohne dass Event Functions Vault darauf aufmerksam wird.
- Honeycomb – Beobachtbarkeits-Tool zum Debuggen Ihres App-Stacks.
- Lumigo— Profiliert die Aufrufe der Lambda-Ereignisfunktion und sammelt Metriken zur Behebung von Problemen in serverlosen Umgebungen und Microservice-Umgebungen.
- New Relic— Läuft zusammen mit Lambda Event Functions und sammelt, verbessert und überträgt automatisch Telemetriedaten an die Unified Observability-Plattform von New Relic.
- Sedai— Eine autonome Cloud-Management-Plattform auf Basis von AI/ML, die Cloud-Betriebsteams kontinuierlich optimiert, um Cloud-Kosteneinsparungen, Leistung und Verfügbarkeit im großen Maßstab zu maximieren.
- Sentry— Diagnostizieren, korrigieren und optimieren Sie die Leistung von Lambda-Ereignisfunktionen.

- Site24x7 – Erreichen Sie Echtzeit-Beobachtbarkeit in Ihren Lambda-Umgebungen
- Splunk— Sammelt hochauflösende Metriken mit niedriger Latenz für eine effiziente und effektive Überwachung von Lambda-Ereignisfunktionen.
- Sumo Logic – Bietet Einblick in den Zustand und die Leistung von Serverless-Anwendungen.
- Salt Security — Vereinfacht die API-Status-Governance und die API-Sicherheit für Lambda Event Functions durch automatisiertes Setup und Support für verschiedene Laufzeiten.

AWS verwaltete Erweiterungen

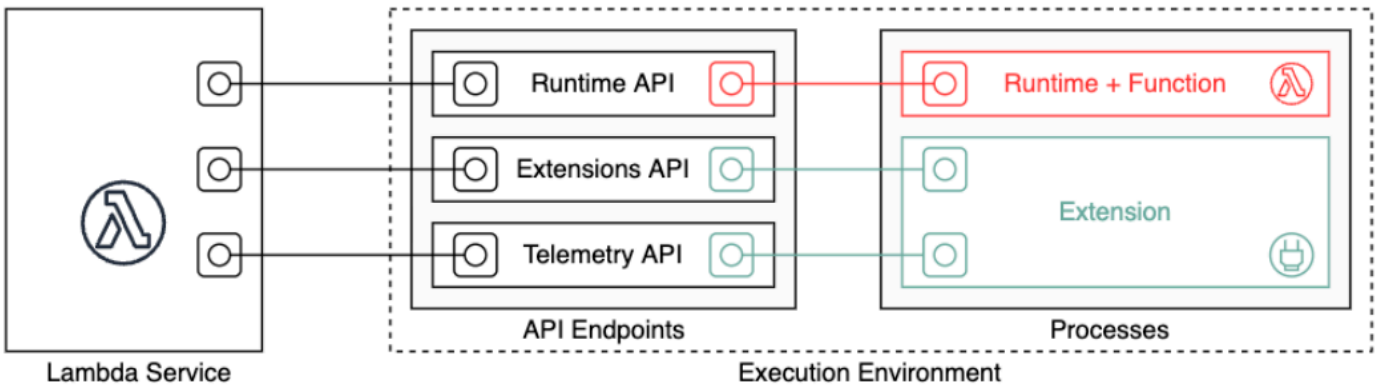
AWS bietet eigene verwaltete Erweiterungen, darunter:

- AWS AppConfig— Verwenden Sie Feature-Flags und dynamische Daten, um Ihre Lambda-Ereignisfunktionen zu aktualisieren. Sie können diese Erweiterung auch verwenden, um andere dynamische Konfigurationen wie Ops-Drosselung und Optimierung zu aktualisieren.
- Amazon CodeGuru Profiler — Verbessert die Anwendungsleistung und senkt die Kosten, indem die teuerste Codezeile einer Anwendung ermittelt und Empfehlungen zur Codeverbesserung gegeben werden.
- CloudWatch Lambda Insights — Überwachen, beheben und optimieren Sie die Leistung Ihrer Lambda Event Functions mithilfe automatisierter Dashboards.
- AWS Distro for OpenTelemetry (ADOT) — Ermöglicht Event Functions das Senden von Trace-Daten an AWS Monitoring-Dienste wie Honeycomb und Lightstep sowie AWS X-Ray an OpenTelemetry unterstützende Ziele wie Honeycomb und Lightstep.
- AWS Parameter und Geheimnisse — Rufen Sie sicher Parameter aus dem AWS Systems Manager Parameterspeicher und Geheimnisse aus AWS Secrets Manager Lambda Event Functions ab.

Weitere Erweiterungs-Beispiele und Demo-Projekte finden Sie unter Erweiterungen für AWS Lambda.

Verwendung der Lambda-Erweiterungs-API zur Erstellung von Erweiterungen

Autoren von Lambda Event Function verwenden Erweiterungen, um Lambda in ihre bevorzugten Tools für Überwachung, Observability, Sicherheit und Governance zu integrieren. Autoren von Event Function können Erweiterungen von AWS, AWS Partnern und Open-Source-Projekten verwenden. Weitere Informationen zur Verwendung von Erweiterungen finden Sie unter Einführung in AWS Lambda Erweiterungen im AWS Compute-Blog. In diesem Abschnitt wird beschrieben, wie Sie die Lambda-Erweiterungs-API, den Lebenszyklus der Lambda-Ausführungsumgebung und die Lambda-Erweiterungs-API-Referenz verwenden.



Als Erweiterungsautor können Sie die Lambda-Erweiterungs-API verwenden, um sich tief in die Lambda-Ausführungsumgebung zu integrieren. Ihre Erweiterung kann sich für Ereignisse im Lebenszyklus der Event-Funktion und der Ausführungsumgebung registrieren. Als Reaktion auf diese Ereignisse können Sie neue Prozesse starten, Logik ausführen und alle Phasen des Lambda-Lebenszyklus steuern und an ihnen teilnehmen: Initialisierung, Aufruf und Herunterfahren. Darüber hinaus können Sie die Laufzeitprotokoll-API verwenden, um einen Stream von Protokollen zu erhalten.

Eine Erweiterung wird als unabhängiger Prozess in der Ausführungsumgebung ausgeführt und kann auch nach der vollständigen Verarbeitung des Aufrufs der Ereignisfunktion weiter ausgeführt werden. Da Erweiterungen als Prozesse ausgeführt werden, können Sie sie in einer anderen Sprache als die Ereignisfunktion schreiben. Es wird empfohlen, Erweiterungen mit einer kompilierten Sprache zu implementieren.

In diesem Fall handelt es sich bei der Erweiterung um eine eigenständige Binärdatei, die mit unterstützten Laufzeiten kompatibel ist. Alle Laufzeiten der Lambda-Event-Funktion unterstützen

Erweiterungen. Wenn Sie eine nicht kompilierte Sprache verwenden, stellen Sie sicher, dass Sie eine kompatible Laufzeit in die Erweiterung aufnehmen.

Lambda unterstützt auch interne Erweiterungen. Eine interne Erweiterung wird als separater Thread im Laufzeitprozess ausgeführt. Die Laufzeit startet und stoppt die interne Erweiterung. Eine alternative Möglichkeit zur Integration in die Lambda-Umgebung ist die Verwendung sprachspezifischer Umgebungsvariablen und Wrapper-Skripte. Sie können diese nutzen, um die Laufzeitumgebung zu konfigurieren und das Startup-Verhalten des Laufzeitprozesses zu verändern.

Sie können einer Ereignisfunktion auf zwei Arten Erweiterungen hinzufügen. Für eine Ereignisfunktion, die als ZIP-Dateiarchiv bereitgestellt wird, stellen Sie Ihre Erweiterung als Ebene bereit. Für eine als Container-Image definierte Event-Funktion fügen Sie die Erweiterungen zu Ihrem Container-Image hinzu.

> **Note**
>
> Beispiele für Erweiterungen und Wrapper-Skripte finden Sie unter AWS Lambda Erweiterungen im AWS GitHub Samples-Repository.

Themen

- Lebenszyklus der Lambda-Ausführungsumgebung
- Erweiterungs-API-Referenz

Lebenszyklus der Lambda-Ausführungsumgebung

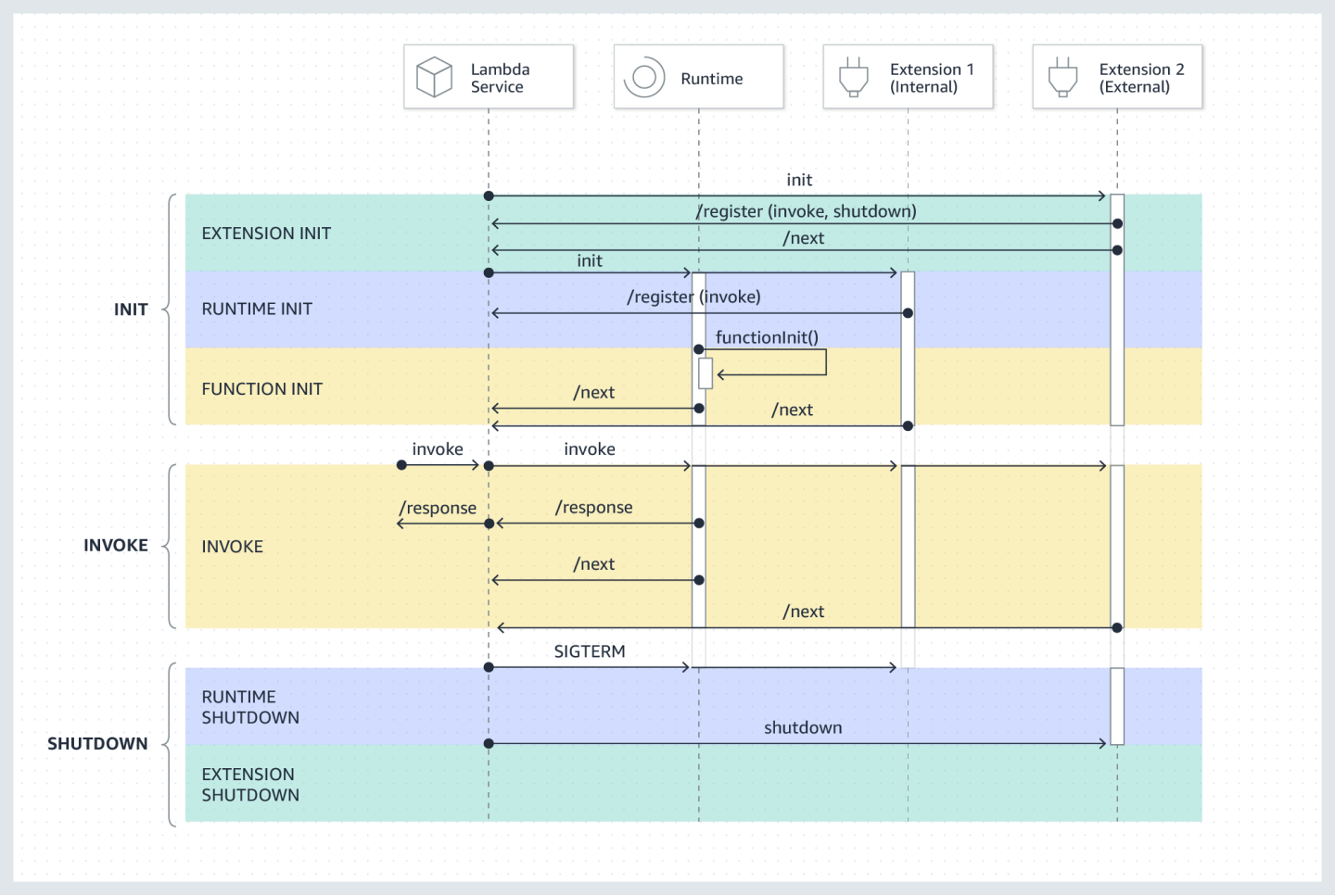
Der Lebenszyklus der Ausführungsumgebung umfasst die folgenden Phasen:

- **`Init`**: In dieser Phase erstellt oder entsperrt Lambda eine Ausführungsumgebung mit den konfigurierten Ressourcen, lädt den Code für die Event-Funktion und alle Ebenen herunter, initialisiert alle Erweiterungen, initialisiert die Laufzeit und führt dann den Initialisierungscode der Event-Funktion aus (den Code außerhalb des Haupthandlers). Die `Init` Phase findet entweder beim ersten Aufruf oder vor den Aufrufen der Event-Funktion statt, wenn Sie die bereitgestellte Parallelität aktiviert haben.

  Die `Init`-Phase ist in drei Unterphasen unterteilt: `Extension init`, `Runtime init` und `Function init`. Diese Unterphasen stellen sicher, dass alle Erweiterungen und die Runtime ihre Einrichtungsaufgaben abschließen, bevor der Event-Function-Code ausgeführt wird.

- **`Invoke`**: In dieser Phase ruft Lambda den Event Function Handler auf. Nachdem die Event-Funktion vollständig ausgeführt wurde, bereitet sich Lambda darauf vor, einen weiteren Event-Funktionsaufruf zu verarbeiten.
- **`Shutdown`**: Diese Phase wird ausgelöst, wenn die Lambda-Ereignisfunktion für einen bestimmten Zeitraum keine Aufrufe empfängt. In dieser `Shutdown`-Phase fährt Lambda die Laufzeit herunter, warnt die Erweiterungen, damit sie sauber beendet werden können und entfernt dann die Umgebung. Lambda sendet ein `Shutdown`-Ereignis an jede Erweiterung, das der Erweiterung mitteilt, dass die Umgebung beendet wird.

Jede Phase beginnt mit einem Ereignis von Lambda zur Laufzeit und zu allen registrierten Erweiterungen. Die Laufzeit und jede Erweiterung zeigen den Abschluss durch Senden einer `Next`-API-Anfrage an. Lambda friert die Ausführungsumgebung ein, wenn jeder Prozess abgeschlossen ist und keine ausstehenden Ereignisse vorhanden sind.



Themen

- Init-Phase
- Invoke-Phase
- Shutdown-Phase
- Berechtigungen und Konfiguration
- Fehlerbehandlung
- Fehlerbehebung bei Erweiterungen

Init-Phase

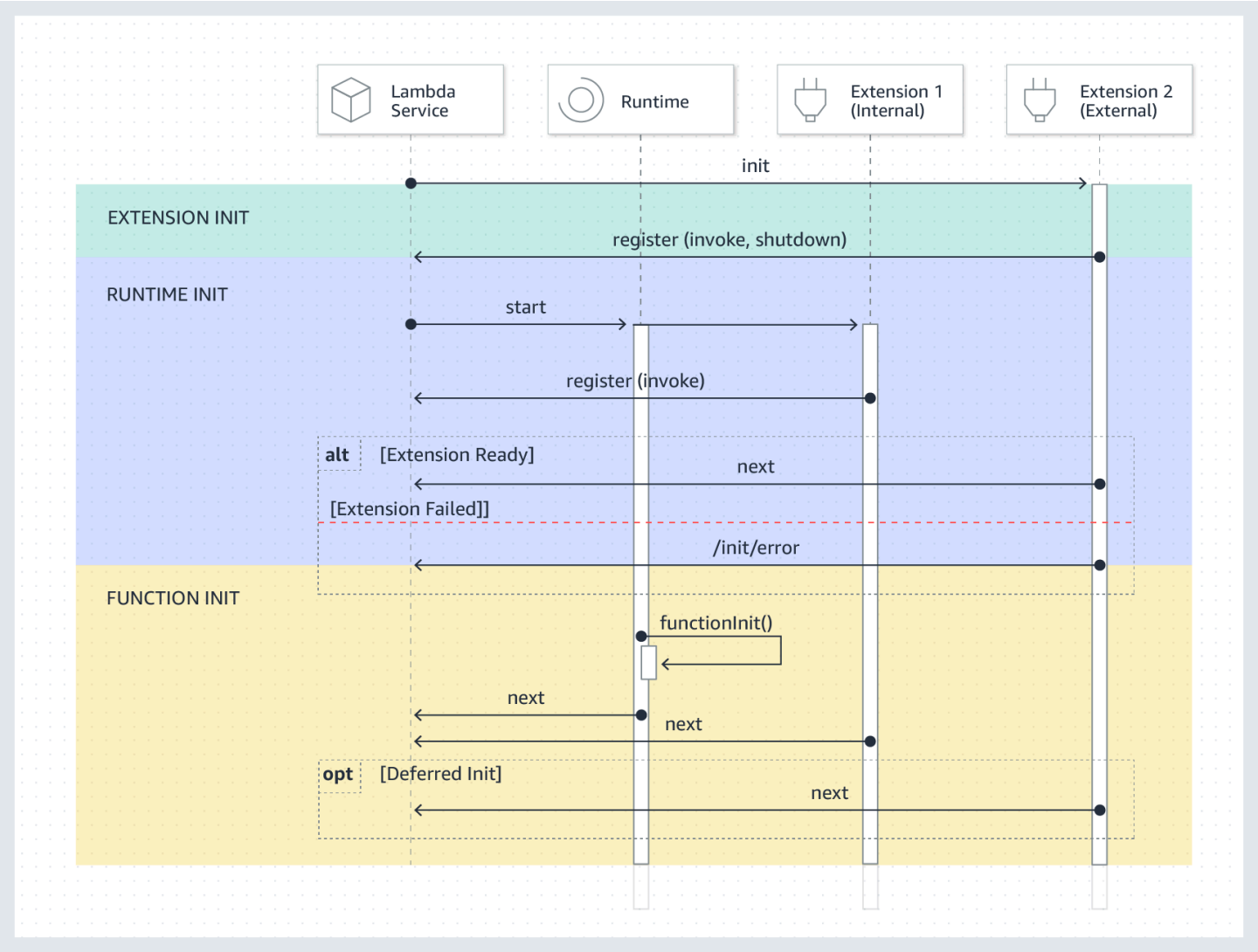
Während dieser `Extension init`-Phase muss sich jede Erweiterung bei Lambda registrieren, um Ereignisse zu empfangen. Lambda verwendet den vollständigen Dateinamen der Erweiterung, um zu überprüfen, ob die Erweiterung die Bootstrap-Sequenz abgeschlossen hat. Daher muss jeder `Register`-API-Aufruf den `Lambda-Extension-Name`-Header mit dem vollständigen Dateinamen der Erweiterung enthalten.

Sie können bis zu 10 Erweiterungen für eine Event-Funktion registrieren. Dieses Limit wird durch den `Register`-API-Aufruf erzwungen.

Nachdem jede Erweiterung registriert ist, startet Lambda die `Runtime init`-Phase. Der Laufzeitprozess ruft `functionInit` auf, um die `Function init`-Phase zu starten.

Die `Init`-Phase wird nach der Laufzeit abgeschlossen und jede registrierte Erweiterung zeigt den Abschluss durch Senden einer `Next`-API-Anforderung an.

> **Note**
>
> Erweiterungen können ihre Initialisierung an jedem beliebigen Punkt in der `Init`-Phase abschließen.



Invoke-Phase

Wenn eine Lambda-Ereignisfunktion als Antwort auf eine `Next` API-Anfrage aufgerufen wird, sendet Lambda ein `Invoke` Ereignis an die Laufzeit und an jede Erweiterung, die für das Ereignis registriert ist. `Invoke`

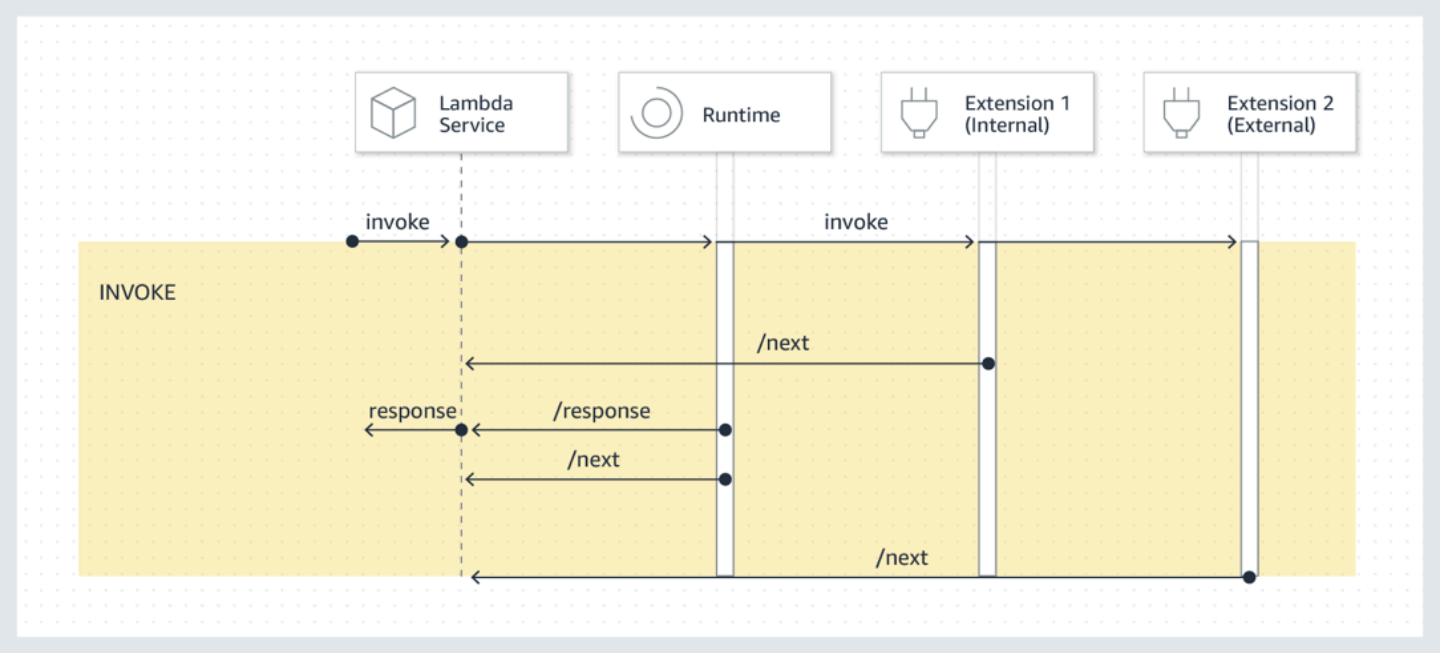
> **Note**
>
> Lambda Managed Instances: Erweiterungen für Funktionen von Lambda Managed Instances können sich nicht für das `Invoke` Ereignis registrieren. Da Lambda Managed Instances gleichzeitige Aufrufe innerhalb einer einzigen Ausführungsumgebung unterstützt, wird das `Invoke` Ereignis nicht unterstützt. Erweiterungen können sich nur für das Ereignis

registrieren. `Shutdown` Wenn Sie nachverfolgen möchten, wann Aufrufe beginnen und enden, verwenden Sie das `platform.report` Plattformereignis über die Telemetrie-API.

Während des Aufrufs werden externe Erweiterungen parallel zur Event-Funktion ausgeführt. Sie laufen auch weiter, nachdem die Event-Funktion abgeschlossen ist. Sie können damit Diagnoseinformationen erfassen oder Protokolle, Messwerte und Traces an einen Ort Ihrer Wahl senden.

Nachdem Lambda die Antwort der Event-Funktion von der Laufzeit erhalten hat, gibt es die Antwort an den Client zurück, auch wenn Erweiterungen noch laufen.

Die `Invoke`-Phase endet nach der Laufzeit und alle Erweiterungen signalisieren durch Senden einer `Next`-API-Anforderung, dass sie ausgeführt wurden.



Event-Payload: Das an die Runtime (und die Lambda-Event-Funktion) gesendete Ereignis trägt die gesamte Anfrage, Header (z. B.`RequestId`) und Nutzdaten. Das an jede Erweiterung gesendete Ereignis enthält Metadaten, die den Ereignisinhalt beschreiben. Dieses Lebenszyklusereignis umfasst den Typ des Ereignisses, den Zeitpunkt, zu dem die Event-Funktion das Timeout (`deadlineMs`) überschreitet ()`requestId`, den Amazon-Ressourcennamen (ARN) der aufgerufenen Event-Funktion und die Tracing-Header.

Erweiterungen, die auf den Ereignistext der Event-Funktion zugreifen möchten, können ein SDK zur Laufzeit verwenden, das mit der Erweiterung kommuniziert. Entwickler von Ereignisfunktionen

verwenden das SDK zur Laufzeit, um die Nutzdaten an die Erweiterung zu senden, wenn die Ereignisfunktion aufgerufen wird.

Hier ist ein Beispiel für eine Nutzlast:

```
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

Zeitlimit: Die Timeout-Einstellung der Event-Funktion begrenzt die Dauer der gesamten Phase. `Invoke` Wenn Sie beispielsweise das Timeout für die Event-Funktion auf 360 Sekunden festlegen, müssen die Event-Funktion und alle Erweiterungen innerhalb von 360 Sekunden abgeschlossen sein. Beachten Sie, dass es keine unabhängige Post-Invoke-Phase gibt. Die Dauer ist die Gesamtzeit, die Sie benötigen, bis Ihre Laufzeit und alle Aufrufe Ihrer Erweiterungen abgeschlossen sind. Sie wird erst berechnet, wenn die Ausführung der Ereignisfunktion und aller Erweiterungen abgeschlossen ist.

Auswirkungen auf die Leistung und der Mehraufwand für Erweiterungen: Erweiterungen können sich auf die Leistung der Event-Funktion auswirken. Als Autor einer Erweiterung haben Sie die Kontrolle über die Auswirkungen Ihrer Erweiterung auf die Leistung. Wenn Ihre Erweiterung beispielsweise rechenintensive Operationen ausführt, verlängert sich die Dauer der Ereignisfunktion, da sich die Erweiterung und der Code der Ereignisfunktion dieselben CPU-Ressourcen teilen. Wenn Ihre Erweiterung nach Abschluss des Aufrufs der Ereignisfunktion umfangreiche Operationen ausführt, erhöht sich außerdem die Dauer der Ereignisfunktion, da die `Invoke` Phase so lange andauert, bis alle Erweiterungen signalisieren, dass sie abgeschlossen sind.

> **Note**
>
> Lambda weist CPU-Leistung proportional zur Speichereinstellung der Event-Funktion zu. Bei niedrigeren Speichereinstellungen kann es zu einer längeren Ausführungs- und Initialisierungsdauer kommen, da die Ereignisfunktion und die Erweiterungsprozesse um

dieselben CPU-Ressourcen konkurrieren. Erhöhen Sie die Speichereinstellung, um die Ausführungs- und Initialisierungsdauer zu reduzieren.

Um die Leistungsbeeinträchtigung zu identifizieren, die durch Erweiterungen in der `Invoke`-Phase verursacht wird, gibt Lambda die `PostRuntimeExtensionsDuration`-Metrik aus. Diese Metrik misst die kumulative Zeit, die zwischen der `Next`-Laufzeit-API-Anforderung und der letzten `Next`-Erweiterungs-API-Anforderung vergeht. Verwenden Sie die `MaxMemoryUsed`-Metrik, um die Zunahme des verwendeten Speichers zu messen. Weitere Informationen zu Metriken für Event-Funktionen finden Sie unterCloudWatch Metriken mit Lambda verwenden.

Entwickler von Ereignisfunktionen können verschiedene Versionen ihrer Ereignisfunktionen nebeneinander ausführen, um die Auswirkungen einer bestimmten Erweiterung zu verstehen. Wir empfehlen den Autoren von Erweiterungen, den voraussichtlichen Ressourcenverbrauch zu veröffentlichen, um Entwicklern von Event Function die Auswahl einer geeigneten Erweiterung zu erleichtern.

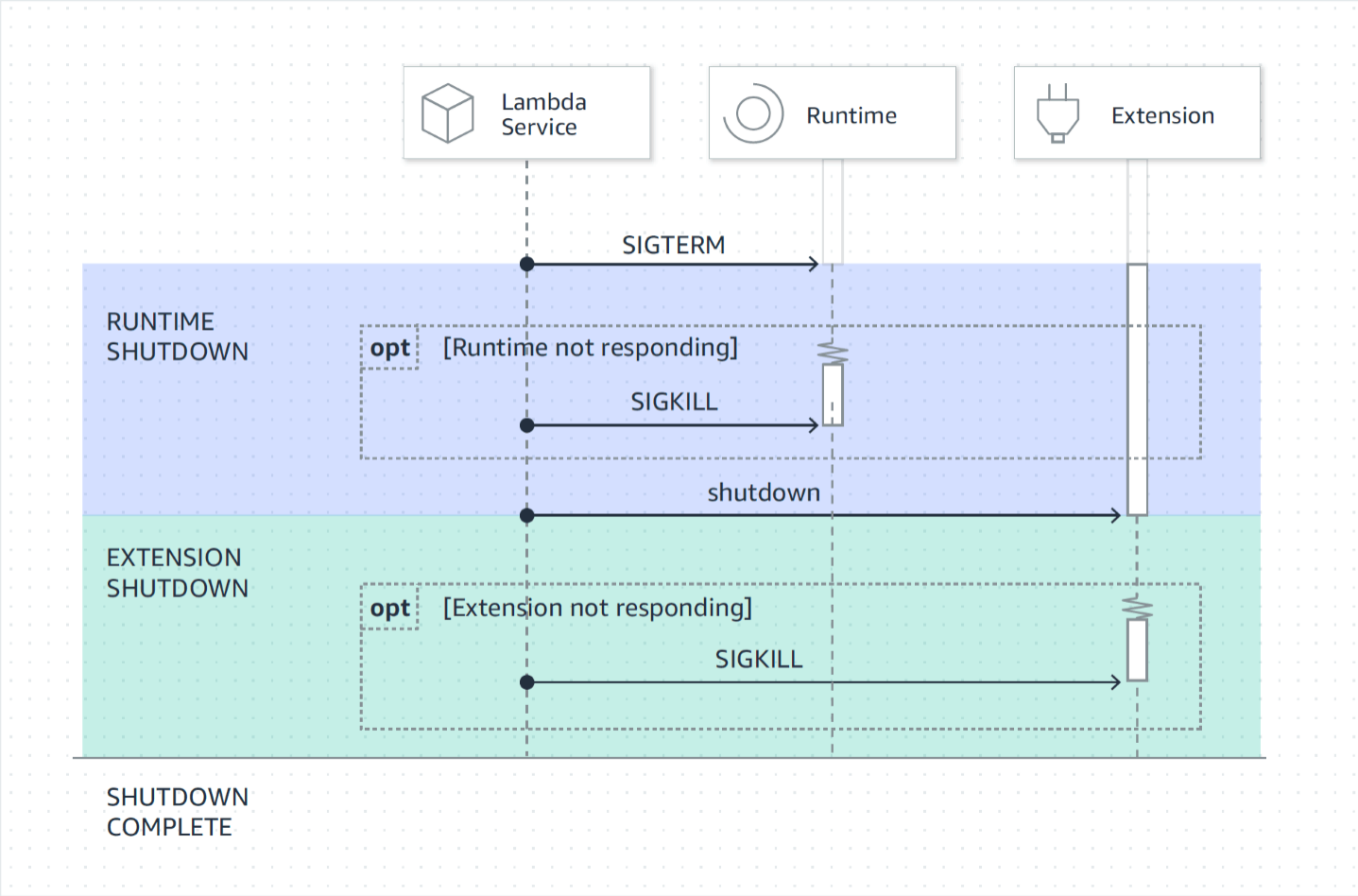
Shutdown-Phase

Wenn Lambda dabei ist, die Laufzeit zu beenden, sendet es ein `Shutdown` an die Laufzeitumgebung und dann an jede registrierte externe Erweiterung. Erweiterungen können diese Zeit für abschließende Bereinigungsaufgaben verwenden. Das `Shutdown`-Ereignis wird als Antwort auf eine `Next`-API-Anforderung gesendet.

Duration Limit (Begrenzung der Dauer): Die maximale Dauer der `Shutdown`-Phase hängt von der Konfiguration der registrierten Erweiterungen ab:

- 0 ms – Eine Funktion ohne registrierte Erweiterungen
- 500 ms – Eine Funktion mit einer registrierten internen Erweiterung
- 2.000 ms – Eine Funktion mit einer oder mehreren registrierten externen Erweiterungen

Wenn die Laufzeit oder eine Erweiterung nicht innerhalb des Limits auf das `Shutdown`-Ereignis reagiert, beendet Lambda den Prozess mit einem `SIGKILL`-Signal.



Event Payload (Ereignis-Nutzlast): Das `Shutdown`-Ereignis enthält den Grund für das Abschalten und die verbleibende Zeit in Millisekunden.

Das `shutdownReason` beinhaltet die folgenden Werte:

- SPINDOWN – Normale Abschaltung
- TIMEOUT – Zeitlimit abgelaufen
- FAILURE – Fehlerzustand, etwa ein `out-of-memory`-Ereignis

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

Berechtigungen und Konfiguration

Erweiterungen werden in derselben Ausführungsumgebung wie die Lambda-Ereignisfunktion ausgeführt. Erweiterungen teilen sich mit der Event-Funktion auch Ressourcen wie CPU, Arbeitsspeicher und `/tmp` Festplattenspeicher. Darüber hinaus verwenden Erweiterungen dieselbe AWS Identity and Access Management (IAM-) Rolle und denselben Sicherheitskontext wie die Event-Funktion.

Dateisystem- und Netzwerkzugriffsberechtigungen: Erweiterungen werden im selben Dateisystem und Netzwerknamensraum ausgeführt wie die Event Function-Laufzeit. Dies bedeutet, dass Erweiterungen mit dem zugehörigen Betriebssystem kompatibel sein müssen. Wenn für eine Erweiterung zusätzliche Regeln für ausgehenden Netzwerkverkehr erforderlich sind, müssen Sie diese Regeln auf die Konfiguration der Ereignisfunktion anwenden.

> **Note**
>
> Da das Verzeichnis mit dem Code der Ereignisfunktion schreibgeschützt ist, können Erweiterungen den Code der Ereignisfunktion nicht ändern.

Umgebungsvariablen: Erweiterungen können auf die Umgebungsvariablen der Ereignisfunktion zugreifen, mit Ausnahme der folgenden Variablen, die für den Laufzeitprozess spezifisch sind:

- `AWS_EXECUTION_ENV`
- `AWS_LAMBDA_LOG_GROUP_NAME`
- `AWS_LAMBDA_LOG_STREAM_NAME`
- `AWS_XRAY_CONTEXT_MISSING`
- `AWS_XRAY_DAEMON_ADDRESS`
- `LAMBDA_RUNTIME_DIR`
- `LAMBDA_TASK_ROOT`
- `_AWS_XRAY_DAEMON_ADDRESS`
- `_AWS_XRAY_DAEMON_PORT`
- `_HANDLER`

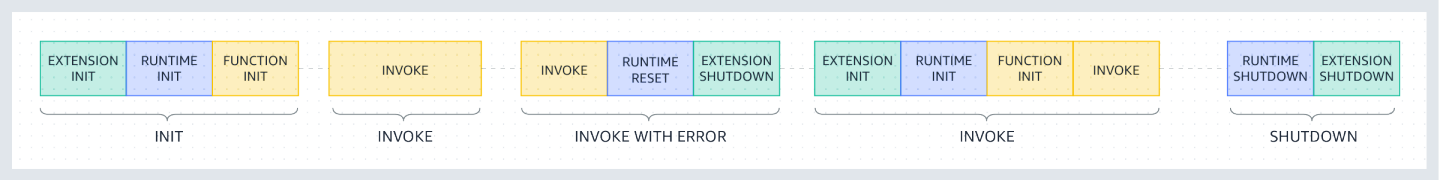
> **Note**
>
> Erkennung von Lambda Managed Instances: Erweiterungen können anhand der `AWS_LAMBDA_INITIALIZATION_TYPE` Umgebungsvariablen überprüfen, ob sie auf Lambda Managed Instances oder auf Lambda-Funktionen (Standard) ausgeführt werden. Dies ist die empfohlene Methode für Erweiterungen, um ihr Verhalten an den Typ der Ausführungsumgebung anzupassen.

Fehlerbehandlung

Initialization failures (Initialisierungsfehler): Wenn eine Erweiterung fehlschlägt, startet Lambda die Ausführungsumgebung neu, um konsistentes Verhalten zu erzwingen und Fail Fast für Erweiterungen zu unterstützen. Für einige Kunden müssen die Erweiterungen auch geschäftskritische Anforderungen wie Protokollierung, Sicherheit, Governance und Telemetrieerfassung erfüllen.

Fehler beim Aufrufen (z. B. zu wenig Arbeitsspeicher, Timeout bei Eventfunktionen): Da Erweiterungen Ressourcen gemeinsam mit der Laufzeit nutzen, wirkt sich dies auf sie aus. Wenn die Laufzeit ausfällt, nehmen alle Erweiterungen und die Laufzeit selbst an der `Shutdown`-Phase teil. Darüber hinaus wird die Laufzeit neu gestartet — entweder automatisch als Teil des aktuellen Aufrufs oder durch einen Mechanismus zur verzögerten Neuinitialisierung.

Wenn währenddessen ein Fehler auftritt (z. B. ein Timeout für die Event-Funktion oder ein Laufzeitfehler)`Invoke`, führt der Lambda-Service einen Reset durch. Der Reset verhält sich wie ein `Shutdown`-Ereignis. Zuerst beendet Lambda die Laufzeit und sendet dann ein `Shutdown`-Ereignis an jede registrierte externe Erweiterung. Das Ereignis enthält den Grund für das Abschalten. Wenn diese Umgebung für einen neuen Aufruf verwendet wird, werden die Erweiterung und die Laufzeit als Teil des nächsten Aufrufers neu initialisiert.



Eine ausführlichere Erläuterung des vorherigen Diagramms finden Sie unter Fehler während der Aufrufphase.

Erweiterungsprotokolle: Lambda sendet die Protokollausgabe von Erweiterungen an CloudWatch Logs. Lambda generiert auch ein zusätzliches Protokollereignis für jede Erweiterung während `Init`.

Das Protokollereignis zeichnet den Namen und die Registrierungseinstellung (Ereignis, Konfiguration) bei Erfolg oder die Fehlerursache bei einem Fehler auf.

Fehlerbehebung bei Erweiterungen

- Wenn eine `Register`-Anforderung fehlschlägt, stellen Sie sicher, dass der `Lambda-Extension-Name`-Header im `Register`-API-Aufruf den vollständigen Dateinamen der Erweiterung enthält.
- Wenn die `Register`-Anforderung für eine interne Erweiterung fehlschlägt, stellen Sie sicher, dass sich die Anforderung nicht für das `Shutdown`-Ereignis registriert.

Erweiterungs-API-Referenz

Die OpenAPI-Spezifikation für die Erweiterungs-API Version 2020-01-01 finden Sie hier: extensions-api.zip

Sie können den Wert des API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariablen abrufen. Um eine `Register`-Anforderung zu senden, verwenden Sie das Präfix `2020-01-01/` vor jedem API-Pfad. Zum Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API-Methoden

- Registrieren
- Next
- Init-Fehler
- Exit-Fehler

Registrieren

Während `Extension init` müssen sich alle Erweiterungen bei Lambda registrieren, um Ereignisse zu empfangen. Lambda verwendet den vollständigen Dateinamen der Erweiterung, um zu überprüfen, ob die Erweiterung die Bootstrap-Sequenz abgeschlossen hat. Daher muss jeder `Register`-API-Aufruf den `Lambda-Extension-Name`-Header mit dem vollständigen Dateinamen der Erweiterung enthalten.

Interne Erweiterungen werden vom Laufzeitprozess gestartet und gestoppt, sodass sie sich nicht für das `Shutdown`-Ereignis registrieren können.

Pfad – `/extension/register`

Methode – POST

Anfordern von Headern

- `Lambda-Extension-Name` – Der vollständige Dateiname der Erweiterung. Erforderlich: ja Typ: Zeichenkette
- `Lambda-Extension-Accept-Feature` – Verwenden Sie dies, um optionale Erweiterungsfunktionen bei der Registrierung anzugeben. Erforderlich: Nein. Typ: durch Kommas getrennte Zeichenfolge. Funktionen, die mit dieser Einstellung angegeben werden können:
  - `accountId`— Falls angegeben, enthält die Antwort auf die Erweiterungsregistrierung die Konto-ID, die der Lambda-Ereignisfunktion zugeordnet ist, für die Sie die Erweiterung registrieren.

Anfrage von Textparametern

- `events` – Array der Ereignisse, für die die Registrierung durchgeführt werden soll. Erforderlich: Nein Typ: Zeichenfolge-Array Gültige Zeichenfolgen: `INVOKE`, `SHUTDOWN`.

> **Note**
>
> Lambda Managed Instances: Erweiterungen für Funktionen von Lambda Managed Instances können sich nur für das `SHUTDOWN` Ereignis registrieren. Der Versuch, sich für das `INVOKE` Ereignis zu registrieren, führt zu einem Fehler. Dies liegt daran, dass Lambda Managed Instances gleichzeitige Aufrufe innerhalb einer einzigen Ausführungsumgebung unterstützt.

Antwort-Header

- `Lambda-Extension-Identifier` – Generierte eindeutige Agent-ID (UUID-Zeichenfolge), die für alle nachfolgenden Anforderungen erforderlich ist.

Antwortcodes

- 200 – Antworttext enthält den Funktionsnamen, die Funktionsversion und den Namen des Handlers.
- 400 – Ungültige Anfrage

- 403 – Verboten
- 500 — Container-Fehler. Non-recoverable Bundesstaat. Die Erweiterung sollte umgehend beendet werden.

Example Beispielanfragetext

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example Beispielantworttext

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example Beispiel-Antworttext mit optionalem AccountID-Feature

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

Next

Erweiterungen senden eine `Next`-API-Anforderung für den Empfang des nächsten Ereignisses, bei dem es sich um ein `Invoke`-Ereignis oder ein `Shutdown`-Ereignis handeln kann. Der Antworttext enthält die Nutzlast, bei der es sich um ein JSON-Dokument handelt, das Ereignisdaten enthält.

Die Erweiterung sendet eine `Next`-API-Anforderung, um zu signalisieren, dass sie bereit ist, neue Ereignisse zu empfangen. Das ist ein blockierender Aufruf.

Legen Sie für den GET-Aufruf kein Timeout fest, da die Erweiterung für einen bestimmten Zeitraum angehalten werden kann, bis ein Ereignis zurückgegeben werden soll.

Pfad – `/extension/event/next`

Methode – GET

Anfordern von Headern

- `Lambda-Extension-Identifier` – Eindeutiger Bezeichner für die Erweiterung (UUID-Zeichenfolge). Erforderlich: ja Typ: UUID-Zeichenfolge

Antwort-Header

- `Lambda-Extension-Event-Identifier` – Eindeutiger Bezeichner für das Ereignis (UUID-Zeichenfolge).

Antwortcodes

- 200 – Die Antwort enthält Informationen über das nächste Ereignis (`EventInvoke` oder `EventShutdown`).
- 403 – Verboten
- 500 — Container-Fehler. Non-recoverable Bundesstaat. Die Erweiterung sollte umgehend beendet werden.

Init-Fehler

Die Erweiterung verwendet diese Methode, um einen Initialisierungsfehler an Lambda zu melden. Rufen Sie sie auf, wenn die Erweiterung nach der Registrierung nicht initialisiert werden kann. Nachdem der Fehler von Lambda empfangen wurde, sind keine weiteren API-Aufrufe erfolgreich. Die Erweiterung sollte beendet werden, nachdem sie die Antwort von Lambda erhalten hat.

Pfad – `/extension/init/error`

Methode – POST

Anfordern von Headern

- `Lambda-Extension-Identifier` – Eindeutiger Bezeichner für die Erweiterung. Erforderlich: ja Typ: UUID-Zeichenfolge
- `Lambda-Extension-Function-Error-Type` – Der Fehlertyp, auf den die Erweiterung gestoßen ist. Erforderlich: ja Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:

- Extension.NoSuchHandler
- Extension.APIKeyNotFound
- Extension.ConfigInvalid
- Extension.UnknownReason

Anfrage von Textparametern

- `ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antwortcodes

- 202 – Akzeptiert
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 — Container-Fehler. Non-recoverable Bundesstaat. Die Erweiterung sollte umgehend beendet werden.

Exit-Fehler

Die Erweiterung verwendet diese Methode, um Lambda vor dem Beenden einen Fehler zu melden. Rufen Sie sie auf, wenn ein unerwarteter Fehler auftritt. Nachdem der Fehler von Lambda empfangen wurde, sind keine weiteren API-Aufrufe erfolgreich. Die Erweiterung sollte beendet werden, nachdem sie die Antwort von Lambda erhalten hat.

Pfad – `/extension/exit/error`

Methode – POST

Anfordern von Headern

- `Lambda-Extension-Identifier` – Eindeutiger Bezeichner für die Erweiterung. Erforderlich: ja Typ: UUID-Zeichenfolge
- `Lambda-Extension-Function-Error-Type` – Der Fehlertyp, auf den die Erweiterung gestoßen ist. Erforderlich: ja Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

Anfrage von Textparametern

- `ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antwortcodes

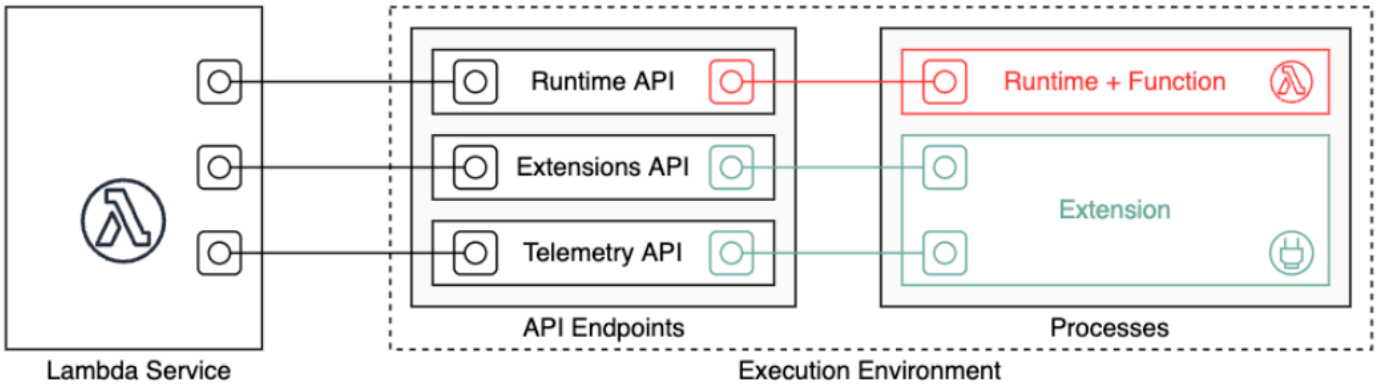
- 202 – Akzeptiert
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 — Container-Fehler. Non-recoverable Bundesstaat. Die Erweiterung sollte umgehend beendet werden.

Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API

Über die Telemetry API können Ihre Erweiterungen Telemetriedaten direkt von Lambda empfangen. Während der Initialisierung und des Aufrufs von Funktionen erfasst Lambda automatisch Telemetriedaten wie Protokolle, Plattform-Metriken und Plattform-Ablaufverfolgungen. Über die Telemetry API können Erweiterungen diese Telemetriedaten direkt von Lambda in nahezu Echtzeit abrufen.

Sie können die Telemetrie-Streams für Ihre Lambda-Erweiterungen direkt in der Lambda-Ausführungsumgebung abonnieren. Nach dem Abonnement sendet Lambda automatisch alle Telemetriedaten an Ihre Erweiterungen. Sie können diese Daten dann verarbeiten, filtern und an Ihr bevorzugtes Ziel senden, z. B. einen Amazon Simple Storage Service (Amazon S3)-Bucket oder einen Drittanbieter von Beobachtbarkeits-Tools.

Das folgende Diagramm zeigt, wie die Extension API und die Telemetry API Erweiterungen von der Ausführungsumgebung aus mit Lambda verbinden. Darüber hinaus verbindet die Runtime-API Ihre Laufzeit- und Event-Funktion mit Lambda.



Important

Die Lambda-Telemetrie-API ersetzt die Lambda-Protokoll-API. Die Protokoll-API bleibt zwar voll funktionsfähig, wir empfehlen jedoch, in Zukunft nur die Telemetrie-API zu verwenden. Sie können Ihre Erweiterung für einen Telemetrie-Stream entweder über die Telemetrie-API oder die Protokoll-API abonnieren. Nach dem Abonnieren mithilfe einer dieser APIs gibt jeder Versuch, über die andere API zu abonnieren, einen Fehler zurück.

Anforderung an die Schemaversion von Lambda Managed Instances

Lambda Managed Instances unterstützen nur die `2025-01-29` Schemaversion der Telemetrie-API. Wenn Sie Telemetriestreams für Funktionen der verwalteten Instanz abonnieren, müssen Sie dies `"schemaVersion": "2025-01-29"` in Ihrer Abonnementanfrage verwenden. Die Verwendung früherer Schemaversionen führt dazu, dass Ereignisse von Lambda zurückgewiesen werden.
Die `2025-01-29` Schemaversion ist abwärtskompatibel und kann sowohl mit Lambda Managed Instances als auch mit Lambda-Funktionen (Standard) verwendet werden. Wir empfehlen, diese Version für alle neuen Erweiterungen zu verwenden, um die Kompatibilität zwischen beiden Bereitstellungsmodellen sicherzustellen.

Erweiterungen können die Telemetrie-API verwenden, um drei verschiedene Telemetrie-Streams zu abonnieren:

- Plattformtelemetrie – Protokolle, Metriken und Ablaufverfolgungen, die Ereignisse und Fehler im Zusammenhang mit dem Laufzeitlebenszyklus der Ausführungsumgebung, dem Erweiterungslebenszyklus und Funktionsaufrufen beschreiben.
- Funktionsprotokolle — Benutzerdefinierte Protokolle, die der Lambda Event Function-Code generiert.
- Erweiterungsprotokolle – Benutzerdefinierte Protokolle, die der Lambda-Erweiterungscode generiert.

Note

Lambda sendet Logs und Metriken sowie Traces an X-Ray (sofern Sie Tracing aktiviert haben), auch wenn eine Erweiterung Telemetriestreams abonniert. CloudWatch

Sections

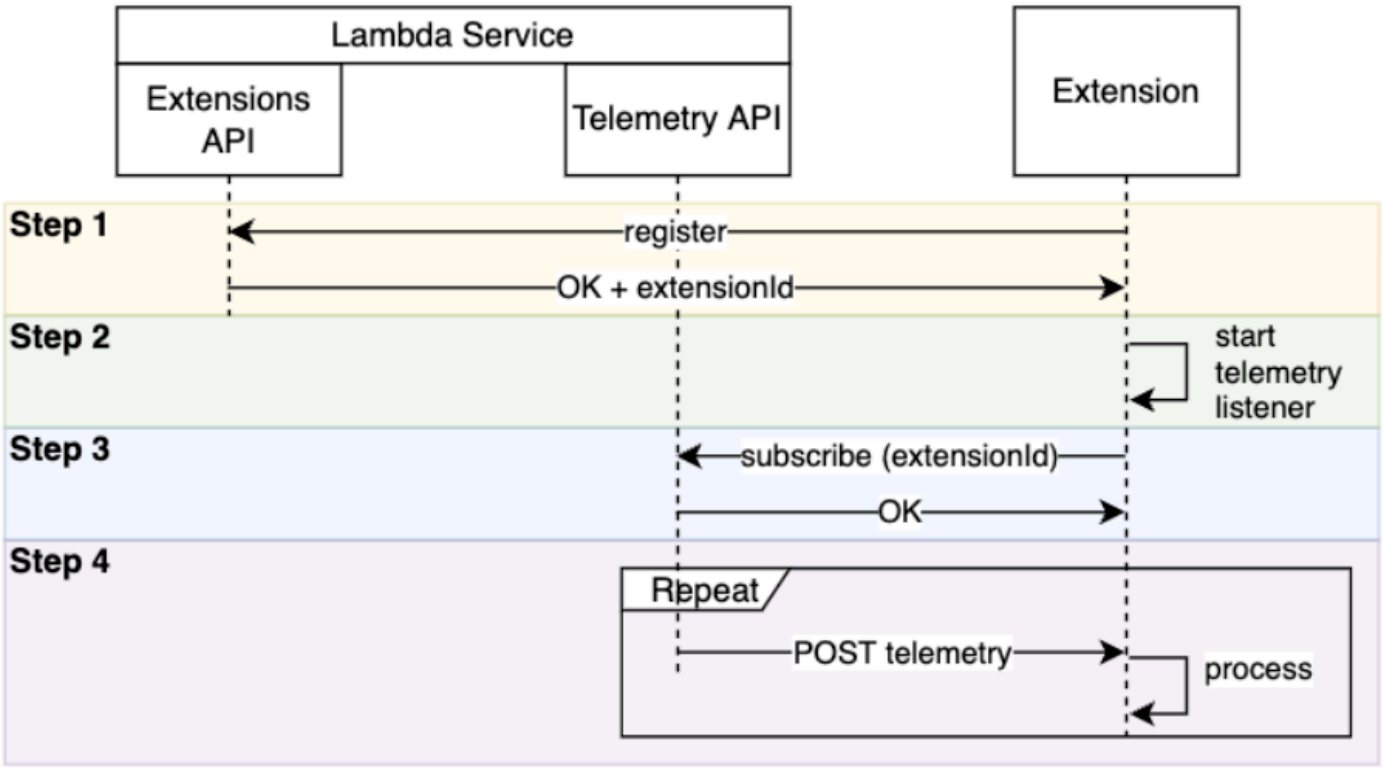
- Erstellen von Erweiterungen mithilfe der Telemetrie-API
- Registrieren Ihrer Erweiterung
- Erstellen eines Telemetrie-Listeners
- Festlegen eines Zielprotokolls

- Konfiguration der Speichernutzung und Pufferung
- Senden einer Abonnementanfrage an die Telemetrie-API
- Eingehende Telemetrie-API-Nachrichten
- Referenz zur Lambda-Telemetrie-API
- Lambda-Telemetrie-API-Ereignisschemareferenz
- Lambda-Telemetrie-API-Ereignisobjekte in Spans konvertieren OpenTelemetry
- Verwendung der Lambda-Protokoll-API

Erstellen von Erweiterungen mithilfe der Telemetrie-API

Lambda-Erweiterungen werden als unabhängige Prozesse in der Ausführungsumgebung ausgeführt. Erweiterungen können nach Abschluss des Funktionsaufrufs weiter ausgeführt werden. Da Erweiterungen separate Prozesse sind, können Sie diese in einer anderen Sprache als dem Funktionscode schreiben. Es wird empfohlen, Erweiterungen mit einer kompilierten Sprache wie Golang oder Rust zu schreiben. Auf diese Weise ist die Erweiterung eine in sich geschlossene Binärdatei, die mit jeder unterstützten Laufzeitumgebung kompatibel sein kann.

Das folgende Diagramm zeigt einen vierstufigen Prozess zum Erstellen einer Erweiterung, die Telemetriedaten mithilfe der Telemetrie-API empfängt und verarbeitet.



Hier sind die einzelnen Schritte im Detail:

1. Registrieren Sie Ihre Erweiterung mithilfe von the section called “Erweiterungs-API”. Dadurch erhalten Sie einen `Lambda-Extension-Identifier`, den Sie in den folgenden Schritten benötigen. Weitere Informationen zum Registrieren einer Erweiterung finden Sie unter the section called “Registrieren Ihrer Erweiterung”.
2. Erstellen eines Telemetrie-Listeners. Dies kann ein einfacher HTTP- oder TCP-Server sein. Lambda verwendet den URI des Telemetrie-Listeners, um Telemetriedaten an Ihre Erweiterung zu senden. Weitere Informationen finden Sie unter the section called “Erstellen eines Telemetrie-Listeners”.
3. Verwenden Sie die Subscribe API in der Telemetry API, um die gewünschten Telemetrie-Streams für Ihre Erweiterung zu abonnieren. Für diesen Schritt benötigen Sie den URI Ihres Telemetrie-Listeners. Weitere Informationen finden Sie unter the section called “Senden einer Abonnementanfrage an die Telemetrie-API”.
4. Rufen Sie Telemetriedaten von Lambda über den Telemetrielistener ab. Sie können diese Daten beliebig weiterverarbeiten, z. B. die Daten an Amazon S3 oder an einen externen Beobachtbarkeits-Service weiterleiten.

> **Note**
>
> Die Ausführungsumgebung einer Event-Funktion kann im Rahmen ihres Lebenszyklus mehrmals gestartet und gestoppt werden. Im Allgemeinen wird Ihr Erweiterungscode bei Aufrufen der Ereignisfunktion und auch während der Shutdown-Phase bis zu 2 Sekunden lang ausgeführt. Wir empfehlen, Telemetriedaten zu Batches zusammenzufassen, sobald sie bei Ihrem Listener eingehen. Verwenden Sie dann die Lebenszyklusereignisse `Invoke` und `Shutdown`, um jeden Batch an die gewünschten Ziele zu senden.

Registrieren Ihrer Erweiterung

Bevor Sie Telemetriedaten abonnieren können, müssen Sie Ihre Lambda-Erweiterung registrieren. Die Registrierung erfolgt während der Initialisierungsphase der Erweiterung. Das folgende Beispiel zeigt eine HTTP-Anfrage zur Registrierung einer Erweiterung.

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Wenn die Anfrage erfolgreich ist, erhält der Subscriber eine Erfolgsantwort von HTTP 200. Der Antwort-Header enthält den `Lambda-Extension-Identifier`. Der Antworttext enthält weitere Eigenschaften der Funktion.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

Weitere Informationen hierzu finden Sie unter the section called “Erweiterungs-API-Referenz”.

Erstellen eines Telemetrie-Listeners

Ihre Lambda-Erweiterung muss über einen Listener verfügen, der eingehende Anfragen der Telemetrie-API verarbeitet. Der folgende Code zeigt eine Beispielimplementierung eines Telemetrie-Listeners in Golang:

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}

// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)
```

```
 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }

 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

Festlegen eines Zielprotokolls

Wenn Sie den Empfang von Telemetrie mithilfe der Telemetrie-API abonnieren, können Sie zusätzlich zur Ziel-URI ein Zielprotokoll angeben:

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda akzeptiert zwei Protokolle für den Empfang von Telemetrie:

- HTTP (empfohlen) – Lambda übermittelt Telemetrie an einen lokalen HTTP-Endpunkt (`http://sandbox.localdomain:${PORT}/${PATH}`) als ein Array von Datensätzen im JSON-Format. Der Parameter `$PATH` ist optional. Lambda-unterstützt nur HTTP, nicht HTTPS. Lambda übermittelt Telemetrie über POST-Anfragen.
- TCP – Lambda übermittelt Telemtrie an einen TCP-Anschluss im Newline-delimited-JSON-(NDJSON)-Format.

> **Note**
>
> Wir empfehlen dringend, HTTP und nicht TCP zu verwenden. Mit TCP kann die Lambda-Plattform nicht bestätigen, dass es Telemetrie an die Anwendungsebene übermittelt. Wenn Ihre Erweiterung abstürzt, verlieren Sie möglicherweise Telemetriedaten. Bei HTTP gibt es diese Einschränkung nicht.

Bevor Sie sich für den Empfang von Telemetriedaten registrieren, richten Sie den lokalen HTTP-Listener oder TCP-Port ein. Beachten Sie bei der Einrichtung Folgendes:

- Lambda sendet Telemetriedaten nur an Ziele, die sich innerhalb der Ausführungsumgebung befinden.
- Lambda wiederholt den Versuch, die Telemetriedaten (mit Backoff) zu senden, wenn es keinen Listener gibt oder wenn die POST-Anfrage zu einem Fehler führt. Wenn der Telemetrie-Listener abstürzt, erhält er nach dem Neustart der Ausführungsumgebung durch Lambda wieder Telemetriedaten.
- Lambda reserviert Port 9001. Es gibt keine weiteren Einschränkungen oder Empfehlungen für Portnummern.

Konfiguration der Speichernutzung und Pufferung

Der Speicherverbrauch in einer Ausführungsumgebung wächst linear mit der Anzahl der Abonnenten. Abonnements verbrauchen Speicherressourcen, da jedes einen neuen Speicherpuffer zum Speichern von Telemetriedaten öffnet. Die Pufferspeichernutzung wird zum Gesamtspeicherverbrauch in der Ausführungsumgebung gezählt.

Wenn Sie den Empfang von Telemetriedaten mithilfe der Telemetry API abonnieren, können Sie Telemetriedaten puffern und sie in Batches an Abonnenten übermitteln. Um die Speichernutzung zu optimieren, können Sie eine Pufferungskonfiguration angeben:

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

Parameter	Description	Standardwerte und Limits
`maxBytes`	Das maximale Telemetri evolumen (in Bytes), das im Speicher gepuffert werden soll.	Standard: 262 144 Mindestwert: 262 144

Parameter	Description	Standardwerte und Limits
		Höchstwert: 1 048 576
`maxItems`	Die maximale Anzahl von Ereignissen, die im Speicher gepuffert werden sollen.	Standard: 10 000 Mindestwert 1 000 Höchstwert: 10 000.
`timeoutMs`	Die maximale Zeit (in Millisekunden) zum Puffern eines Batches.	Standard: 1 000 Minimum: 25 Höchstwert: 30 000

Beachten Sie beim Einrichten der Pufferung die folgenden Punkte:

- Wenn einer der Eingabestreams geschlossen wird, leert Lambda die Protokolle. Dies kann beispielsweise passieren, wenn die Laufzeit abstürzt.
- Jeder Subscriber kann in seiner Abonnementanfrage eine andere Pufferungskonfiguration angeben.
- Rechnen Sie beim Bestimmen der Puffergröße für das Lesen der Daten mit eingehenden Nutzlasten in der Größenordnung von `2 * maxBytes + metadataBytes`, wobei `maxBytes` ein Bestandteil Ihrer Pufferungskonfiguration ist. Anhand der folgenden Metadaten können Sie `metadataBytes` abschätzen. Lambda hängt jedem Datensatz ähnliche Metadaten an:

```
{
   "time": "2022-08-20T12:31:32.123Z",
   "type": "function",
   "record": "Hello World"
}
```

- Wenn der Abonnent eingehende Telemetrie nicht schnell genug verarbeiten kann oder wenn Ihr Event-Function-Code ein sehr hohes Protokollvolumen generiert, kann Lambda Datensätze löschen, um die Speicherauslastung zu begrenzen. In diesem Fall sendet Lambda ein `platform.logsDropped`-Ereignis.

Senden einer Abonnementanfrage an die Telemetrie-API

Lambda-Erweiterungen können den Empfang von Telemetriedaten abonnieren, indem sie eine Abonnementanfrage an die Telemetrie-API senden. Die Abonnementanfrage sollte Informationen über die Arten von Ereignissen enthalten, die die Erweiterung abonnieren soll. Darüber hinaus kann die Anfrage Informationen zum Lieferziel und eine Pufferkonfiguration enthalten.

Bevor Sie eine Abonnementanfrage senden, benötigen Sie eine Erweiterungs-ID (`Lambda-Extension-Identifier`). Wenn Sie Ihre Erweiterung in der Erweiterungs-API registrieren, erhalten Sie eine Erweiterungs-ID aus der API-Antwort.

Das Abonnement erfolgt während der Initialisierungsphase der Erweiterung. Das folgende Beispiel zeigt eine HTTP-Anfrage zum Abonnieren aller drei Telemetriestreams: Plattformtelemetrie, Funktionsprotokolle und Erweiterungsprotokolle.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

Ist die Anfrage erfolgreich, erhält der Subscriber eine HTTP 200-Erfolgsantwort.

```
HTTP/1.1 200 OK
"OK"
```

Eingehende Telemetrie-API-Nachrichten

Nach dem Abonnieren über die Telemetrie-API beginnt eine Erweiterung automatisch, Telemetrie von Lambda über POST-Anfragen zu empfangen. Jeder POST-Anforderungstext enthält ein Array von `Event` Objekten. Jedes `Event` hat das folgende Schema:

```
{
   time: String,
   type: String,
   record: Object
}
```

- Die `time`-Eigenschaft definiert, wann die Lambda-Plattform das Ereignis generiert hat. Das ist nicht identisch mit dem Zeitpunkt, zu dem das Ereignis tatsächlich stattgefunden hat. Der Zeichenfolgenwert von `time` ist ein Zeitstempel im ISO 8601-Format.
- Die `type`-Eigenschaft definiert den Ereignistyp. Die folgende Tabelle beschreibt alle möglichen Werte.
- Die `record`-Eigenschaft definiert ein JSON-Objekt, das die Telemetriedaten enthält. Das Schema dieses JSON-Objekts hängt von dem `type` ab.

> **Bestellung von Ereignissen mit gleichzeitigen Aufrufen**
>
> Bei Lambda Managed Instances können mehrere Funktionsaufrufen gleichzeitig in derselben Ausführungsumgebung ausgeführt werden. In diesem Fall ist die Reihenfolge der `platform.report` Ereignisse `platform.start` und zwischen verschiedenen gleichzeitigen Aufrufen nicht garantiert. Erweiterungen müssen Ereignisse aus mehreren Aufrufen verarbeiten, die parallel ausgeführt werden, und sollten keine sequentielle Reihenfolge annehmen.
>
> Um Ereignisse bestimmten Aufrufen korrekt zuzuordnen, sollten Erweiterungen das in diesen Plattformereignissen vorhandene `requestId` Feld verwenden. Jeder Aufruf hat eine eindeutige Anforderungs-ID, die für alle Ereignisse dieses Aufrufs einheitlich bleibt, sodass Erweiterungen Ereignisse auch dann korrekt korrelieren können, wenn sie nicht in der richtigen Reihenfolge ankommen.

Die folgende Tabelle fasst alle `Event`-Objekttypen zusammen und enthält Links zur `Event`-Telemetrie-API-Schemareferenz für jeden Ereignistyp.

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.initStart`	Die Funktionsinitialisierung wurde gestartet.	the section called "`Platform.InitStart`"-Schema
Plattform-Ereignis	`platform.initRuntimeDone`	Die Funktionsinitialisierung ist abgeschlossen.	the section called "`platform.initRuntimeDone`"-Schema
Plattform-Ereignis	`platform.initReport`	Ein Bericht über die Funktionsinitialisierung.	the section called "`Platform.InitReport`"-Schema
Plattform-Ereignis	`platform.start`	Der Funktionsaufruf wurde gestartet.	the section called "`Plattform.Start`"-Schema
Plattform-Ereignis	`platform.runtimeDone`	Die Laufzeit hat die Verarbeitung eines Ereignisses erfolgreich oder mit einem Fehler abgeschlossen.	the section called "`platform.runtimeDone`"-Schema
Plattform-Ereignis	`platform.report`	Ein Bericht über den Funktionsaufruf.	the section called "`platform.report`"-Schema
Plattform-Ereignis	`platform.restoreStart`	Die Laufzeitwiederherstellung wurde gestartet.	the section called "`Plattform. Neustart wiederherstellen`"-Schema

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.restoreRuntimeDone`	Die Laufzeitwiederherstellung wurde abgeschlossen.	the section called “`platform.wiederherstellen RuntimeDone`”-Schema
Plattform-Ereignis	`platform.restoreReport`	Bericht über die Laufzeitwiederherstellung.	the section called “`Platform.RestoreReport`”-Schema
Plattform-Ereignis	`platform.telemetrySubscription`	Die Erweiterung hat die Telemetrie-API abonniert.	the section called “`Plattform.Telemetrieabonnement`”-Schema
Plattform-Ereignis	`platform.logsDropped`	Lambda hat die Protokolleinträge gelöscht.	the section called “`Platform.Logs wurde gelöscht`”-Schema
Funktionsprotokolle	`function`	Eine Protokollzeile aus dem Funktionscode.	the section called “`function`”-Schema
Erweiterungsprotokolle	`extension`	Eine Protokollzeile aus dem Erweiterungscode.	the section called “`extension`”-Schema

Referenz zur Lambda-Telemetrie-API

Verwenden Sie den Lambda-Telemetrie-API-Endpunkt, um Erweiterungen für Telemetrie-Streams zu abonnieren. Sie können den Telemtrie-API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariable abrufen. Um eine API-Anfrage zu senden, fügen Sie die API-Version (`2022-07-01/`) und `telemetry/` an. Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

Die Definition der OpenAPI-Spezifikation (OAS) der Abonnementantwortversion `2025-01-29` finden Sie unter:

- HTTP — telemetry-api-http-schema.zip
- TCP — .zip telemetry-api-tcp-schema

API-Operationen

- Abonnieren

Abonnieren

Um einen Telemetrie-Stream zu abonnieren, kann eine Lambda-Erweiterung eine Abonnement-API-Anfrage senden.

- Pfad – `/telemetry`
- Methode – PUT
- Header
  - `Content-Type`: `application/json`
- Anfrage von Textparametern
  - schemaVersion
    - Erforderlich: Ja
    - Typ: Zeichenfolge
    - Gültige Werte: `"2025-01-29"`, `"2022-12-13"` oder `"2022-07-01"`
    - Hinweis: Für Lambda Managed Instances ist Folgendes erforderlich`"2025-01-29"`. Diese Version ist abwärtskompatibel mit Lambda-Funktionen (Standard).

- Ziel – Die Konfigurationseinstellungen, die das Ziel des Telemetrieereignisses und das Protokoll für die Ereignisübermittlung definieren.
  - Erforderlich: Ja
  - Typ: Objekt

```
{
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
}
```

  - Protokoll – Das Protokoll, das Lambda zum Senden von Telemetriedaten verwendet.
    - Erforderlich: Ja
    - Typ: Zeichenfolge
    - Gültige Werte: `"HTTP"|"TCP"`
  - URI – Der URI, an den Telemetriedaten gesendet werden sollen.
    - Erforderlich: Ja
    - Typ: Zeichenfolge
  - Weitere Informationen finden Sie unter the section called “Festlegen eines Zielprotokolls”.
- Typen – Die Telemetrietypen, die die Erweiterung abonnieren soll.
  - Erforderlich: Ja
  - Typ: Zeichenfolgen-Array
  - Zulässige Werte: `"platform"|"function"|"extension"`
- Pufferung – Die Konfigurationseinstellungen für die Ereignispufferung.
  - Erforderlich: Nein
  - Typ: Objekt

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - maxItems – Die maximale Anzahl der Ereignisse im Speicher, die gepuffert werden sollen.

    - Erforderlich: Nein
    - Typ: Ganzzahl
    - Standard: 1 000
    - Mindestwert 1 000
    - Höchstwert: 10 000.
- maxBytes – Das maximale Telemetrievolumen (in Bytes), das im Speicher gepuffert werden soll.
    - Erforderlich: Nein
    - Typ: Ganzzahl
    - Standard: 262 144
    - Mindestwert: 262 144
    - Höchstwert: 1 048 576
- timeoutMs – Die maximale Zeit (in Millisekunden) zum Puffern eines Batches.
    - Erforderlich: Nein
    - Typ: Ganzzahl
    - Standard: 1 000
    - Minimum: 25
    - Höchstwert: 30 000
- Weitere Informationen finden Sie unter the section called “Konfiguration der Speichernutzung und Pufferung”.

Beispiel für eine Abonnement-API-Anfrage

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
```

```
    },
    "destination": {
         "protocol": "HTTP",
         "URI": "http://sandbox.localdomain:8080"
    }
}
```

Wenn die Abonnement-Anfrage erfolgreich war, erhält die Erweiterung eine Erfolgsbestätigung (HTTP 200):

```
HTTP/1.1 200 OK
"OK"
```

Wenn die Abonnement-Anfrage fehlschlägt, erhält die Erweiterung eine Fehlerantwort. Beispiel:

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

Hier sind einige zusätzliche Antwortcodes, die die Erweiterung empfangen kann:

- 200 – Anfrage erfolgreich abgeschlossen
- 202 – Anfrage wurde akzeptiert. Antwort auf die Abonnement-Anfrage in einer lokalen Testumgebung
- 400 – Ungültige Anfrage
- 500 – Servicefehler

Lambda-Telemetrie-API-Ereignisschemareferenz

Verwenden Sie den Lambda-Telemetrie-API-Endpunkt, um Erweiterungen für Telemetrie-Streams zu abonnieren. Sie können den Telemtrie-API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariable abrufen. Um eine API-Anfrage zu senden, fügen Sie die API-Version (`2022-07-01/`) und `telemetry/` an. Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

Die Definition der OpenAPI-Spezifikation (OAS) der Abonnementantwortversion `2025-01-29` finden Sie unter:

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

Die folgende Tabelle enthält eine Zusammenfassung aller `Event`-Objekttypen, die die Telemetrie-API unterstützt.

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.initStart`	Die Funktionsinitialisierung wurde gestartet.	the section called “`Platform.InitStart`”-Schema
Plattform-Ereignis	`platform.initRuntimeDone`	Die Funktionsinitialisierung ist abgeschlossen.	the section called “`platform.initRuntimeDone`”-Schema
Plattform-Ereignis	`platform.initReport`	Ein Bericht über die Funktionsinitialisierung.	the section called “`Platform.InitReport`”-Schema

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.start`	Der Funktionsaufruf wurde gestartet.	the section called “`Plattform.Start`”-Schema
Plattform-Ereignis	`platform.runtimeDone`	Die Laufzeit hat die Verarbeitung eines Ereignisses erfolgreich oder mit einem Fehler abgeschlossen.	the section called “`platform.runtimeDone`”-Schema
Plattform-Ereignis	`platform.report`	Ein Bericht über den Funktionsaufruf.	the section called “`platform.report`”-Schema
Plattform-Ereignis	`platform.restoreStart`	Die Laufzeitwiederherstellung wurde gestartet.	the section called “`Plattform. Neustart wiederherstellen`”-Schema
Plattform-Ereignis	`platform.restoreRuntimeDone`	Die Laufzeitwiederherstellung wurde abgeschlossen.	the section called “`plattform.wiederherstellen RuntimeDone`”-Schema
Plattform-Ereignis	`platform.restoreReport`	Bericht über die Laufzeitwiederherstellung.	the section called “`Platform.RestoreReport`”-Schema

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.telemetrySubscription`	Die Erweiterung hat die Telemetrie-API abonniert.	the section called “`Plattform.Telemetrieabonnement`”-Schema
Plattform-Ereignis	`platform.logsDropped`	Lambda hat die Protokolleinträge gelöscht.	the section called “`Platform.Logs wurde gelöscht`”-Schema
Funktionsprotokolle	`function`	Eine Protokollzeile aus dem Funktionscode.	the section called “`function`”-Schema
Erweiterungsprotokolle	`extension`	Eine Protokollzeile aus dem Erweiterungscode.	the section called “`extension`”-Schema

Inhalt

- Objekttypen für Telemetrie-API-Ereignisse
  - Platform.InitStart
  - platform.init RuntimeDone
  - Platform.InitReport
  - Plattform.Start
  - platform.runtimeDone
  - platform.report
  - Plattform. Neustart wiederherstellen
  - plattform.wiederherstellen RuntimeDone
  - Platform.RestoreReport
  - Plattform.Erweiterung

  - Plattform.Telemetrieabonnement
  - Platform.Logs wurde gelöscht
  - function
  - extension
- Freigegebene Objekttypen
  - InitPhase
  - InitReportMetrics
  - InitType
  - ReportMetrics
  - RestoreReportMetrics
  - RuntimeDoneMetrics
  - Span
  - Status
  - TraceContext
  - TracingType

Objekttypen für Telemetrie-API-Ereignisse

Dieser Abschnitt beschreibt die Arten der `Event`-Objekte, die die Lambda-Telemetrie-API unterstützt. In den Ereignisbeschreibungen weist ein Fragezeichen (?) darauf hin, dass das Attribut möglicherweise nicht in diesem Objekt vorhanden ist.

Platform.InitStart

Ein `platform.initStart` Ereignis zeigt an, dass die Initialisierungsphase der Event-Funktion gestartet wurde. Ein `platform.initStart` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

Dieses `PlatformInitStart`-Objekt hat die folgenden Attribute:

- functionName – `String`

- functionVersion – `String`
- initializationType – the section called “InitType”-Objekt
- instanceId? – `String`
- Instanz? MaxMemory — `Integer`
- phase – the section called “InitPhase”-Objekt
- runtimeVersion? – `String`
- LaufzeitVersionArn? — `String`

Es folgt ein Beispiel für `Event` des Typs `platform.initStart`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

`platform.init RuntimeDone`

Ein `platform.initRuntimeDone` Ereignis zeigt an, dass die Initialisierungsphase der Event-Funktion abgeschlossen ist. Ein `platform.initRuntimeDone` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

Dieses `PlatformInitRuntimeDone`-Objekt hat die folgenden Attribute:

- initializationType – the section called “InitType”-Objekt

- phase – the section called “InitPhase”-Objekt
- status – the section called “Status”-Objekt
- spans? – Liste von the section called “Span”-Objekten

Es folgt ein Beispiel für `Event` des Typs `platform.initRuntimeDone`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

`Platform.InitReport`

Ein `platform.initReport` Ereignis enthält einen Gesamtbericht über die Initialisierungsphase der Event-Funktion. Ein `platform.initReport` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

Dieses `PlatformInitReport`-Objekt hat die folgenden Attribute:

- errorType? – Zeichenfolge
- initializationType – the section called “InitType”-Objekt
- phase – the section called “InitPhase”-Objekt
- metrics – the section called “InitReportMetrics”-Objekt

- spans? – Liste von the section called “Span”-Objekten
- status – the section called “Status”-Objekt

Es folgt ein Beispiel für `Event` des Typs `platform.initReport`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
}
```

`Plattform.Start`

Ein `platform.start` Ereignis zeigt an, dass die Aufrufphase der Event-Funktion gestartet wurde. Ein `platform.start` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

Dieses `PlatformStart`-Objekt hat die folgenden Attribute:

- requestId – `String`
- version? – `String`
- tracing? – the section called “TraceContext”

Es folgt ein Beispiel für `Event` des Typs `platform.start`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "version": "$LATEST",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

`platform.runtimeDone`

Ein `platform.runtimeDone` Ereignis zeigt an, dass die Aufrufphase der Event-Funktion abgeschlossen ist. Ein `platform.runtimeDone` `Event`-Objekt hat die folgende Form:

> **Von Lambda verwaltete Instances**
>
> Das `platform.runtimeDone` Ereignis wird für Lambda Managed Instances nicht unterstützt. Erweiterungen, die auf verwalteten Instanzen ausgeführt werden, erhalten dieses Ereignis nicht, da Erweiterungen das `INVOKE` Ereignis auf verwalteten Instanzen nicht abonnieren können. Aufgrund des Modells der gleichzeitigen Ausführung, bei dem mehrere Aufrufe gleichzeitig verarbeitet werden können, können Erweiterungen keine Verarbeitung nach dem Aufrufen für einzelne Aufrufe durchführen, wie dies üblicherweise bei Lambda-Funktionen (Standard) der Fall ist.
> Bei verwalteten Instanzen sind die Bereiche `responseLatency` und, die normalerweise `responseDuration` enthalten sind, stattdessen im Ereignis verfügbar. `platform.runtimeDone` `platform.report` Details dazu finden Sie unter the section called “`platform.report`”.

```
Event: Object
- time: String
```

```
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

Dieses `PlatformRuntimeDone`-Objekt hat die folgenden Attribute:

- errorType? – `String`
- metrics? – the section called “RuntimeDoneMetrics” Objekt
- requestId – `String`
- status – the section called “Status”-Objekt
- spans? – Liste von the section called “Span”-Objekten
- tracing? – the section called “TraceContext”-Objekt

Es folgt ein Beispiel für `Event` des Typs `platform.runtimeDone`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

`platform.report`

Ein `platform.report` Ereignis enthält einen Gesamtbericht über die Aufrufphase der Ereignisfunktion. Ein `platform.report` `Event`-Objekt hat die folgende Form:

> **Von Lambda verwaltete Instances**
>
> Das `platform.report` Ereignis für Lambda Managed Instances hat im Vergleich zu Lambda-Funktionen (Standard) andere Metriken und Spannweiten. Für verwaltete Instanzen:
>
> - Spans: Enthält `responseLatency` und `responseDuration` statt. `extensionOverhead` Die `extensionOverhead` Spanne ist nicht verfügbar, da Erweiterungen das `INVOKE` Ereignis auf verwalteten Instanzen aufgrund des parallelen Ausführungsmodells nicht abonnieren können.
> - Metriken: Beinhaltet `durationMs` nur. Die folgenden Metriken sind nicht enthalten: `billedDurationMs`,`initDurationMs`,`maxMemoryUsedMB`, und`memorySizeMB`. Diese Metriken pro Aufruf sind in der Umgebung für die gleichzeitige Ausführung nicht anwendbar. Verwenden Überwachung von mit Lambda verwalteten Instances Sie Lambda Insights für Kennzahlen zur Ressourcennutzung.

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

Dieses `PlatformReport`-Objekt hat die folgenden Attribute:

- metrics – `the section called “ReportMetrics”`-Objekt
- requestId – `String`
- spans? – Liste von `the section called “Span”`-Objekten
- Status – `the section called “Status”`-Objekt
- Nachverfolgung? – `the section called “TraceContext”`-Objekt

Es folgt ein Beispiel für `Event` des Typs `platform.report`:

```
{
```

```
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "billedDurationMs": 694,
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

`Plattform. Neustart wiederherstellen`

Ein `platform.restoreStart` Ereignis weist darauf hin, dass ein Ereignis zur Wiederherstellung der Event Function-Umgebung gestartet wurde. Bei einem Wiederherstellungsereignis einer Umgebung erstellt Lambda die Umgebung aus einem im Cache gespeicherten Snapshot, anstatt diese von Grund auf neu zu initialisieren. Weitere Informationen finden Sie unter SnapStart. Ein `platform.restoreStart` `Event` Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

Dieses `PlatformRestoreStart`-Objekt hat die folgenden Attribute:

- functionName – `String`
- functionVersion – `String`
- instanceId? – `String`
- InstanzMaxMemory? — `String`
- runtimeVersion? – `String`
- LaufzeitVersionArn? — `String`

Es folgt ein Beispiel für `Event` des Typs `platform.restoreStart`:

```
{
```

```
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

`plattform.wiederherstellen RuntimeDone`

Ein `platform.restoreRuntimeDone` Ereignis weist darauf hin, dass ein Ereignis zur Wiederherstellung der Event Function-Umgebung abgeschlossen wurde. Bei einem Wiederherstellungsereignis einer Umgebung erstellt Lambda die Umgebung aus einem im Cache gespeicherten Snapshot, anstatt diese von Grund auf neu zu initialisieren. Weitere Informationen finden Sie unter SnapStart. Ein `platform.restoreRuntimeDone` `Event` Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

Dieses `PlatformRestoreRuntimeDone`-Objekt hat die folgenden Attribute:

- errorType? – `String`
- spans? – Liste von `the section called “Span”`-Objekten
- status – `the section called “Status”`-Objekt

Es folgt ein Beispiel für `Event` des Typs `platform.restoreRuntimeDone`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
```

```
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ]
    }
}
```

`Platform.RestoreReport`

Ein `platform.restoreReport` Ereignis enthält einen Gesamtbericht über ein Wiederherstellungsereignis für die Event-Funktion. Ein `platform.restoreReport` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

Dieses `PlatformRestoreReport`-Objekt hat die folgenden Attribute:

- errorType? – Zeichenfolge
- metrics? – the section called “RestoreReportMetrics” Objekt
- spans? – Liste von the section called “Span”-Objekten
- status – the section called “Status”-Objekt

Es folgt ein Beispiel für `Event` des Typs `platform.restoreReport`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
```

```
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
}
```

Plattform.Erweiterung

Ein `extension`-Ereignis enthält Protokolle aus dem Erweiterungscode. Ein `extension` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

Dieses `PlatformExtension`-Objekt hat die folgenden Attribute:

- events – Liste von `String`
- name – `String`
- status – `String`

Es folgt ein Beispiel für `Event` des Typs `platform.extension`:

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

Plattform.Telemetrieabonnement

Ein `platform.telemetrySubscription`-Ereignis enthält Informationen zu einem Erweiterungsabonnement. Ein `platform.telemetrySubscription` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

Dieses `PlatformTelemetrySubscription`-Objekt hat die folgenden Attribute:

- name – `String`
- status – `String`
- types – Liste von `String`

Es folgt ein Beispiel für `Event` des Typs `platform.telemetrySubscription`:

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

`Platform.Logs wurde gelöscht`

Ein `platform.logsDropped`-Ereignis enthält Informationen über gelöschte Ereignisse. Lambda gibt das `platform.logsDropped` Ereignis aus, wenn eine Event-Funktion Logs mit einer zu hohen Rate ausgibt, als dass Lambda sie verarbeiten könnte. Wenn Lambda Protokolle nicht mit der Geschwindigkeit an CloudWatch oder an die Erweiterung senden kann, die die Telemetrie-API abonniert hat, werden Protokolle gelöscht, um zu verhindern, dass die Ausführung der Ereignisfunktion verlangsamt wird. Ein `platform.logsDropped` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

Dieses `PlatformLogsDropped`-Objekt hat die folgenden Attribute:

- droppedBytes – `Integer`
- droppedRecords – `Integer`
- reason – `String`

Es folgt ein Beispiel für `Event` des Typs `platform.logsDropped`:

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

`function`

Ein `function` Ereignis enthält Protokolle aus dem Code der Ereignisfunktion. Ein `function` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = function
- record: {}
```

Das Format des `record` Felds hängt davon ab, ob die Protokolle Ihrer Event-Funktion im Klartext- oder JSON-Format formatiert sind. Weitere Informationen zu den Konfigurationsoptionen für das Protokollformat finden Sie unter the section called “Protokollformate”

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `function`, bei dem das Protokollformat Klartext ist:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `function`, bei dem das Protokollformat JSON ist:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
        "timestamp": "2022-10-12T00:03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

> **Note**
>
> Wenn die von Ihnen verwendete Schemaversion älter als die `2022-12-13` Version ist, wird sie immer als Zeichenfolge gerendert, auch wenn das Protokollierungsformat Ihrer Event-Funktion als JSON konfiguriert ist. `"record"` Für Lambda Managed Instances müssen Sie die Schemaversion `2025-01-29` verwenden.

`extension`

Ein `extension`-Ereignis enthält Protokolle aus dem Erweiterungscode. Ein `extension` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

Das Format des `record` Felds hängt davon ab, ob die Logs Ihrer Event-Funktion im Klartext- oder JSON-Format formatiert sind. Weitere Informationen zu den Konfigurationsoptionen für das Protokollformat finden Sie unter the section called “Protokollformate”

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `extension`, bei dem das Protokollformat Klartext ist:

```
{
```

```
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `extension`, bei dem das Protokollformat JSON ist:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

> **Note**
>
> Wenn die von Ihnen verwendete Schemaversion älter als die `2022-12-13` Version ist, wird sie immer als Zeichenfolge gerendert, auch wenn das Protokollierungsformat Ihrer Event-Funktion als JSON konfiguriert ist. `"record"` Für Lambda Managed Instances müssen Sie die Schemaversion `2025-01-29` verwenden.

Freigegebene Objekttypen

Dieser Abschnitt beschreibt die Arten von freigegebenen Objekten, die die Lambda-Telemetrie-API unterstützt.

`InitPhase`

Eine Zeichenfolgenaufzählung, die die Phase beschreibt, in der der Initialisierungsschritt auftritt. In den meisten Fällen führt Lambda den Initialisierungscode der Event Function während der `init` Phase aus. In einigen Fehlerfällen kann Lambda jedoch den Initialisierungscode der Ereignisfunktion während der Phase erneut ausführen. `invoke` (Dies wird als suppressed init bezeichnet.)

- Typ: – `String`

- Gültige Werte – `init|invoke|snap-start`

`InitReportMetrics`

Ein Objekt, das Metriken zu einer Initialisierungsphase enthält.

- Typ: – `Object`

Ein `InitReportMetrics`-Objekt hat die folgende Form:

```
InitReportMetrics: Object
- durationMs: Double
```

Es folgt ein Beispiel für ein `InitReportMetrics`-Objekt:

```
{
    "durationMs": 247.88
}
```

`InitType`

Eine Zeichenfolgenaufzählung, die beschreibt, wie Lambda die Umgebung initialisiert hat.

- Typ: – `String`
- Gültige Werte – `on-demand|provisioned-concurrency`

`ReportMetrics`

Ein Objekt, das Details zu einer abgeschlossenen Phase enthält.

- Typ: – `Object`

Ein `ReportMetrics`-Objekt hat die folgende Form:

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
```

```
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

Es folgt ein Beispiel für ein `ReportMetrics`-Objekt:

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

`RestoreReportMetrics`

Ein Objekt, das Metriken über eine abgeschlossene Wiederherstellungsphase enthält.

- Typ: – `Object`

Ein `RestoreReportMetrics`-Objekt hat die folgende Form:

```
RestoreReportMetrics: Object
- durationMs: Double
```

Es folgt ein Beispiel für ein `RestoreReportMetrics`-Objekt:

```
{
    "durationMs": 15.19
}
```

`RuntimeDoneMetrics`

Ein Objekt, das Metriken über eine Aufrufphase enthält.

- Typ: – `Object`

Ein `RuntimeDoneMetrics`-Objekt hat die folgende Form:

```
RuntimeDoneMetrics: Object
- durationMs: Double
```

```
- producedBytes?: Integer
```

Es folgt ein Beispiel für ein `RuntimeDoneMetrics`-Objekt:

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

Ein Objekt, das Details einem Span enthält. Ein Span stellt eine Arbeitseinheit oder einen Vorgang in einer Ablaufverfolgung dar. Weitere Informationen zu Spans finden Sie unter Span auf der Tracing-API-Seite der Docs-Website. OpenTelemetry

Lambda unterstützt die folgenden zwei Bereiche für das `platform.RuntimeDone`-Ereignis:

- Die `responseLatency` Spanne beschreibt, wie lange es gedauert hat, bis Ihre Lambda-Event-Funktion mit dem Senden der Antwort begonnen hat.
- Die `responseDuration` Spanne beschreibt, wie lange es gedauert hat, bis Ihre Lambda-Ereignisfunktion das Senden der gesamten Antwort abgeschlossen hat.
- Die `runtimeOverhead` Spanne beschreibt, wie lange es gedauert hat, bis die Lambda-Laufzeit signalisiert hat, dass sie bereit ist, den nächsten Aufruf der Event-Funktion zu verarbeiten. So lange hat die Laufzeit gedauert, bis die API für den nächsten Aufruf aufgerufen wurde, um das nächste Ereignis abzurufen, nachdem Ihre Antwort auf die Event-Funktion zurückgegeben wurde.

Es folgt ein Beispiel für ein Objekt des `responseLatency`-Spans:

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

Status

Ein Objekt, das den Status einer Initialisierungs- oder Aufrufphase beschreibt. Wenn der Status entweder `failure` oder `error` ist, dann enthält das `Status`-Objekt auch ein `errorType`-Feld, das den Fehler beschreibt.

- Typ: – `Object`
- Gültige Statuswerte – `success|failure|error|timeout`

`TraceContext`

Ein Objekt, das die Eigenschaften einer Ablaufverfolgung beschreibt.

- Typ: – `Object`

Ein `TraceContext`-Objekt hat die folgende Form:

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

Es folgt ein Beispiel für ein `TraceContext`-Objekt:

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

`TracingType`

Eine Zeichenfolgenaufzählung, die den Ablaufverfolgungstyp in einem the section called “`TraceContext`”-Objekt beschreibt.

- Typ: – `String`
- Gültige Werte – `X-Amzn-Trace-Id`

`Lambda-Telemetrie-API-Ereignisobjekte` in Spans konvertieren OpenTelemetry

Das AWS Lambda Telemetrie-API-Schema ist semantisch kompatibel mit (oTEL). OpenTelemetry Das bedeutet, dass Sie Ihre AWS Lambda `Event` Telemetrie-API-Objekte in OpenTelemetry (oTEL) -Spans konvertieren können. Beim Konvertieren sollten Sie kein einzelnes `Event`-Objekt einem einzelnen OTel Span zuordnen.

Stattdessen sollten Sie alle drei Ereignisse, die sich auf eine Lebenszyklusphase beziehen, in einem einzigen OTel-Span darstellen. Die `start` `runtimeReport` Ereignisse,, und stellen beispielsweise einen einzelnen Aufruf der Ereignisfunktion dar. `runtimeDone` Stellen Sie alle drei Ereignisse als einen einzigen OTel-Span dar.

Sie können Ihre Ereignisse mithilfe von Span-Ereignissen oder untergeordneten (verschachtelten) Spans konvertieren. Die Tabellen auf dieser Seite beschreiben die Zuordnungen zwischen Telemetrie-API-Schemaeigenschaften und OTel-Span-Eigenschaften für beide Ansätze. Weitere Informationen zu oTEL Spans finden Sie unter Span auf der Tracing API-Seite der OpenTelemetry Docs-Website.

Sections

- Zuordnung zu OTel Spans mit Span-Ereignissen
- Zuordnung zu OTel-Spans mit untergeordneten Spans

Zuordnung zu OTel Spans mit Span-Ereignissen

In den folgenden Tabellen stellt e das Ereignis dar, das von der Telemetrie-Quelle stammt.

Zuordnung der *Start-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert diesen Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time`.
`Span.EndTime`	N/A, weil die Veranstaltung noch nicht abgeschlossen ist.

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Kind`	Setzen Sie diesen Wert auf `Server`.
`Span.Status`	Setzen Sie diesen Wert auf `Unset`.
`Span.TraceId`	Analysieren Sie den in gefundenen AWS X-Ray Header `e.tracing.value` und verwenden Sie dann den `TraceId` Wert.
`Span.ParentId`	Analysieren Sie den X-Ray Header, der in gefunden wurde`e.tracing.value` , und verwenden Sie dann den `Parent` Wert.
`Span.SpanId`	Verwenden Sie `e.tracing.spanId` , falls verfügbar. Generieren Sie andernfalls eine neue `SpanId`.
`Span.SpanContext.TraceState`	N/A für einen X-Ray Trace-Kontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den in gefundenen X-Ray Header `e.tracing.value` und verwenden Sie dann den `Sampled` Wert.
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerd efinierte Werte hinzufügen.

Zuordnen der RuntimeDone *-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert den Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time` aus dem übereinst immenden `*Start`-Ereignis.

OpenTelemetry	Schema der Lambda-Telemetrie-API
	Als alternative Vorgehensweise verwenden Sie `e.time - e.metrics.durationMs` .
`Span.EndTime`	N/A, weil die Veranstaltung noch nicht abgeschlossen ist.
`Span.Kind`	Setzen Sie diesen Wert auf `Server`.
`Span.Status`	Wenn `e.status` nicht gleich `success` ist, dann Wert auf `Error` festlegen. Ansonsten auf `0k` festlegen.
`Span.Events[]`	Verwenden Sie `e.spans[]` .
`Span.Events[i].Name`	Verwenden Sie `e.spans[i].name` .
`Span.Events[i].Time`	Verwenden Sie `e.spans[i].start` .
`Span.TraceId`	Analysieren Sie den in gefundenen AWS X-Ray Header `e.tracing.value` und verwenden Sie dann den `TraceId` Wert.
`Span.ParentId`	Analysieren Sie den X-Ray Header, der in gefunden wurde`e.tracing.value` , und verwenden Sie dann den `Parent` Wert.
`Span.SpanId`	Verwenden Sie dieselbe `SpanId` aus dem `*Start`-Ereignis. Falls nicht verfügbar, verwenden Sie `e.tracing.spanId` oder generieren Sie eine neue `SpanId`.
`Span.SpanContext.TraceState`	N/A für einen X-Ray Trace-Kontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den in gefundenen X-Ray Header `e.tracing.value` und verwenden Sie dann den `Sampled` Wert.

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerdefinierte Werte hinzufügen.

Zuordnung der *Report-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert den Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time` aus dem übereinstimmenden `*Start`-Ereignis. Als alternative Vorgehensweise verwenden Sie `e.time - e.metrics.durationMs` .
`Span.EndTime`	Verwenden Sie `e.time`.
`Span.Kind`	Setzen Sie diesen Wert auf `Server`.
`Span.Status`	Verwenden Sie den gleichen Wert wie das `*RuntimeDone` -Ereignis.
`Span.TraceId`	Analysieren Sie den AWS X-Ray Header, der in gefunden wurde`.tracing.value` , und verwenden Sie dann den `TraceId` Wert.
`Span.ParentId`	Analysieren Sie den X-Ray Header, der in gefunden wurde`.tracing.value` , und verwenden Sie dann den `Parent` Wert.
`Span.SpanId`	Verwenden Sie dieselbe `SpanId` aus dem `*Start`-Ereignis. Falls nicht verfügbar, verwenden Sie `e.tracing.spanId` oder generieren Sie eine neue `SpanId`.

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.SpanContext.TraceState`	N/A für einen X-Ray Trace-Kontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den in gefundenen X-Ray Header `e.tracing.value` und verwenden Sie dann den `Sampled` Wert.
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerdefinierte Werte hinzufügen.

Zuordnung zu OTel-Spans mit untergeordneten Spans

Die folgende Tabelle beschreibt, wie Lambda-Telemetrie-API-Ereignisse in OTel-Spans mit untergeordneten (verschachtelten) Spans für `*RuntimeDone`-Spans konvertiert werden. Informationen zu `*Start`- und `*Report`-Zuordnungen finden Sie in den Tabellen in the section called “Zuordnung zu OTel Spans mit Span-Ereignissen”, da diese für untergeordnete Spans identisch sind. In dieser Tabelle stellt `e` das Ereignis dar, das von der Telemetrie-Quelle stammt.

Zuordnen der RuntimeDone *-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert den Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time` aus dem übereinstimmenden `*Start`-Ereignis. Als alternative Vorgehensweise verwenden Sie `e.time - e.metrics.durationMs` .
`Span.EndTime`	N/A, weil die Veranstaltung noch nicht abgeschlossen ist.
`Span.Kind`	Setzen Sie diesen Wert auf `Server`.
`Span.Status`	Wenn `e.status` nicht gleich `success` ist, dann Wert auf `Error` festlegen.

OpenTelemetry	Schema der Lambda-Telemetrie-API
	Ansonsten auf 0k festlegen.
`Span.TraceId`	Analysieren Sie den in gefundenen AWS X-Ray Header `e.tracing.value` und verwenden Sie dann den `TraceId` Wert.
`Span.ParentId`	Analysieren Sie den X-Ray Header, der in gefunden wurde`e.tracing.value` , und verwenden Sie dann den `Parent` Wert.
`Span.SpanId`	Verwenden Sie dieselbe `SpanId` aus dem `*Start`-Ereignis. Falls nicht verfügbar, verwenden Sie `e.tracing.spanId` oder generieren Sie eine neue `SpanId`.
`Span.SpanContext.TraceState`	N/A für einen X-Ray Trace-Kontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den in gefundenen X-Ray Header `e.tracing.value` und verwenden Sie dann den `Sampled` Wert.
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerd efinierte Werte hinzufügen.
`ChildSpan[i].Name`	Verwenden Sie `e.spans[i].name` .
`ChildSpan[i].StartTime`	Verwenden Sie `e.spans[i].start` .
`ChildSpan[i].EndTime`	Verwenden Sie `e.spans[i].start + e.spans[i].durations` .
`ChildSpan[i].Kind`	Wie bei übergeordnetem `Span.Kind` .
`ChildSpan[i].Status`	Wie bei übergeordnetem `Span.Status` .
`ChildSpan[i].TraceId`	Wie bei übergeordnetem `Span.TraceId` .

OpenTelemetry	Schema der Lambda-Telemetrie-API
`ChildSpan[i].ParentId`	Verwenden Sie die übergeordnete `Span.SpanId` .
`ChildSpan[i].SpanId`	Generieren Sie eine neue `SpanId`.
`ChildSpan[i].SpanContext.TraceState`	N/A für einen X-Ray Trace-Kontext.
`ChildSpan[i].SpanContext.TraceFlags`	Wie bei übergeordnetem `Span.SpanContext.TraceFlags` .

Verwendung der Lambda-Protokoll-API

Important

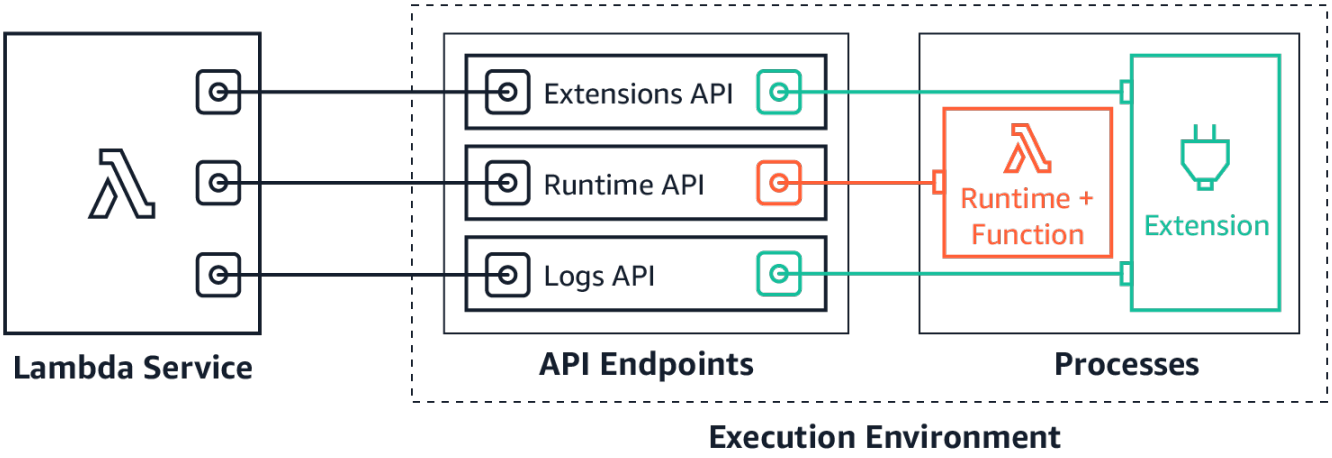
Die Lambda-Telemetrie-API ersetzt die Lambda-Protokoll-API. Die Protokoll-API bleibt zwar voll funktionsfähig, wir empfehlen jedoch, in Zukunft nur die Telemetrie-API zu verwenden. Sie können Ihre Erweiterung für einen Telemetrie-Stream entweder über die Telemetrie-API oder die Protokoll-API abonnieren. Nach dem Abonnieren mithilfe einer dieser APIs gibt jeder Versuch, über die andere API zu abonnieren, einen Fehler zurück.

Lambda Managed Instances unterstützen die Logs-API nicht

Lambda Managed Instances unterstützen die Logs-API nicht. Wenn Sie Funktionen für verwaltete Instanzen verwenden, verwenden Sie stattdessen die Telemetrie-API. Die Telemetrie-API bietet erweiterte Funktionen zum Sammeln und Verarbeiten von Telemetriedaten aus Ihren Lambda-Ereignisfunktionen.

Lambda erfasst automatisch Laufzeitprotokolle und streamt sie an Amazon CloudWatch. Dieser Protokollstream enthält die Protokolle, die Ihr Event-Funktionscode und Ihre Erweiterungen generieren, sowie die Logs, die Lambda als Teil des Event-Funktionsaufrufs generiert.

Lambda-Erweiterungen können die Lambda-Laufzeitprotokoll-API verwenden, um Protokollstreams direkt aus der Lambda-Ausführungsumgebung zu abonnieren. Lambda streamt die Protokolle zur Erweiterung und die Erweiterung kann die Protokolle dann an ein beliebiges bevorzugtes Ziel verarbeiten, filtern und senden.



Die Logs API ermöglicht es Erweiterungen, drei verschiedene Protokollstreams zu abonnieren:

- Die Funktion protokolliert, was die Lambda-Ereignisfunktion generiert und in `stdout` oder `stderr` schreibt.
- Erweiterungsprotokolle, die der Erweiterungscode generiert.
- Lambda-Plattformprotokolle, die Ereignisse und Fehler im Zusammenhang mit Aufrufen und Erweiterungen aufzeichnen.

> **Note**
>
> Lambda sendet alle Protokolle an CloudWatch, auch wenn eine Erweiterung einen oder mehrere der Protokollstreams abonniert.

Themen

- Abonnieren des Empfangs von Protokollen
- Speicherauslastung
- Ziel-Protokolle
- Pufferung der Konfiguration
- Beispielabonnement
- Beispielcode für Logs API
- Logs API-Referenz

- Protokollmeldungen

Abonnieren des Empfangs von Protokollen

Eine Lambda-Erweiterung kann den Empfang von Protokollen abonnieren, indem sie eine Abonnementanfrage an die Protokoll-API sendet.

Um den Empfang von Protokollen zu abonnieren, benötigen Sie die Erweiterungskennung (`Lambda-Extension-Identifier`). Registrieren Sie zunächst die Erweiterung , um die Erweiterungskennung zu erhalten. Abonnieren Sie dann während der Initialisierung die Logs API. Lambda verarbeitet nach Abschluss der Initialisierungsphase keine Abonnementanfragen.

> **Note**
>
> Das Logs API-Abonnement ist idempotent. Doppelte Abonnementanfragen führen nicht zu doppelten Abonnements.

Speicherauslastung

Die Speichernutzung steigt linear, wenn die Anzahl der Abonnenten zunimmt. Abonnements verbrauchen Speicherressourcen, da jedes Abonnement einen neuen Speicherpuffer zum Speichern der Protokolle öffnet. Um die Speicherauslastung zu optimieren, können Sie die Pufferungskonfigurationanpassen. Die Pufferspeichernutzung wird zum Gesamtspeicherverbrauch in der Ausführungsumgebung gezählt.

Ziel-Protokolle

Sie können eines der folgenden Protokolle auswählen, um die Logs zu erhalten:

1. HTTP (empfohlen) – Lambda stellt Protokolle als Array von Datensätzen im JSON-Format an einen lokalen HTTP-Endpunkt (`http://sandbox.localdomain:${PORT}/${PATH}`) bereit. Der Parameter `$PATH` ist optional. Beachten Sie, dass nur HTTP unterstützt wird, nicht HTTPS. Sie können wählen, ob Sie Logs über PUT oder POST erhalten möchten.
2. TCP – Lambda stellt Protokolle an einen TCP-Port im Newline-delimited-JSON-(NDJSON)-Format bereit.

Wir empfehlen, HTTP statt TCP zu verwenden. Mit TCP kann die Lambda-Plattform nicht bestätigen, wenn es Protokolle an die Anwendungsebene übermittelt. Daher können Sie Protokolle verlieren, wenn Ihre Erweiterung abstürzt. Bei HTTP gilt diese Einschränkung nicht.

Wir empfehlen außerdem, den lokalen HTTP-Listener oder den TCP-Port einzurichten, bevor Sie sich für den Empfang von Protokollen anmelden. Beachten Sie bei der Einrichtung Folgendes:

- Lambda sendet Protokolle nur an Ziele, die sich innerhalb der Ausführungsumgebung befinden.
- Lambda wiederholt den Versuch, die Protokolle (mit Backoff) zu senden, wenn es keinen Listener gibt oder wenn die POST- oder PUT-Anfrage zu einem Fehler führt. Wenn der Protokoll-Subscriber abstürzt, erhält er nach dem Neustart der Ausführungsumgebung durch Lambda weiterhin Protokolle.
- Lambda reserviert Port 9001. Es gibt keine weiteren Einschränkungen oder Empfehlungen für Portnummern.

Pufferung der Konfiguration

Lambda kann Protokolle puffern und an den Subscriber liefern. Sie können dieses Verhalten in der Abonnementanfrage konfigurieren, indem Sie die folgenden optionalen Felder angeben. Beachten Sie, dass Lambda den Standardwert für jedes Feld verwenden wird, das Sie nicht angeben.

- timeoutMs – Die maximale Zeit (in Millisekunden) zum Puffern eines Batches. Standard: 1 000. Minimum: 25. Höchstwert: 30 000.
- maxBytes – Die maximale Größe (in Byte) der Protokolle, die im Speicher gepuffert werden sollen. Standard: 262 144. Mindestwert: 262 144. Höchstwert: 1 048 576.
- maxItems – Die maximale Anzahl der Ereignisse im Speicher, die gepuffert werden sollen. Standard: 10 000. Mindestwert 1 000. Höchstwert: 10 000.

Beachten Sie bei der Pufferung die folgenden Punkte:

- Lambda bereinigt die Protokolle, wenn einer der Eingabestreams geschlossen wird, z. B. wenn die Laufzeit abstürzt.
- Jeder Subscriber kann in seiner Abonnementanfrage eine andere Pufferungskonfiguration angeben.
- Berücksichtigen Sie die Puffergröße, die Sie zum Lesen der Daten benötigen. Gehen Sie davon aus, dass die Nutzlasten so groß wie `2*maxBytes+metadata` sein werden, wo `maxBytes` in der

Abonnementanfrage konfiguriert ist. Lambda fügt beispielsweise jedem Datensatz die folgenden Metadaten-Bytes hinzu:

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- Wenn der Subscriber eingehende Protokolle nicht schnell genug verarbeiten kann, wird Lambda Protokolle möglicherweise löschen, um die Speicherauslastung begrenzt zu halten. Um die Anzahl der gelöschten Datensätze anzugeben, schickt Lambda ein `platform.logsDropped`-Protokoll. Weitere Informationen finden Sie unter the section called “Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt”.

Beispielabonnement

Das folgende Beispiel zeigt eine Anfrage zum Abonnieren der Plattform- und Funktionsprotokolle.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

Wenn die Anfrage erfolgreich ist, erhält der Subscriber eine Erfolgsantwort von HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Beispielcode für Logs API

Einen Beispielcode, der zeigt, wie Logs an ein benutzerdefiniertes Ziel gesendet werden, finden Sie im AWS LambdaAWS Compute-Blog unter Verwenden von Erweiterungen zum Senden von Logs an benutzerdefinierte Ziele.

Codebeispiele für Python und Go, die zeigen, wie Sie eine grundlegende Lambda-Erweiterung entwickeln und die Logs-API abonnieren, finden Sie unter AWS Lambda Erweiterungen im AWS GitHub Samples-Repository. Für weitere Informationen zum Erstellen einer Lambda-Erweiterung siehe the section called “Erweiterungs-API”.

Logs API-Referenz

Sie können den Wert des API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariablen abrufen. Um eine API-Anfrage zu senden, verwenden Sie das Präfix `2020-08-15/` vor dem API-Pfad. Zum Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

Die OpenAPI-Spezifikation für die Protokoll-API-Version 2020-08-15, finden Sie hier: logs-api-request.zip

Abonnieren

Um einen oder mehrere der in der Lambda-Ausführungsumgebung verfügbaren Protokollstreams zu abonnieren, senden Erweiterungen eine Abonnement-API-Anforderung.

Pfad – `/logs`

Methode – PUT

Body-Parameter

`destination` – Siehe the section called “Ziel-Protokolle”. Erforderlich: ja Typ: Strings.

`buffering` – Siehe the section called “Pufferung der Konfiguration”. Erforderlich: Nein Typ: Strings.

`types` – Ein Array der zu empfangenden Protokollentypen. Erforderlich: ja Typ: Zeichenfolge-Array Gültige Werte: „platform“, „function“, „extension“.

`schemaVersion` – Erforderlich: Nein. Standardwert: „2020-08-15“. Stellen Sie „2021-03-18“ ein, damit die Erweiterung `platform.runtimeDone`-Meldungen empfangen kann.

Antwortparameter

Die OpenAPI-Spezifikationen für die Abonnementantworten, Version 2020-08-15, stehen für HTTP- und TCP-Protokolle zur Verfügung:

- HTTP: logs-api-http-response.zip
- TCP: logs-api-tcp-response.zip

Antwortcodes

- 200 – Anfrage erfolgreich abgeschlossen
- 202 – Anfrage wurde akzeptiert. Antwort auf eine Abonnementanfrage während lokaler Tests.
- 4XX – Ungültige Anfrage
- 500 – Servicefehler

Wenn die Anfrage erfolgreich ist, erhält der Subscriber eine Erfolgsantwort von HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Schlägt die Anfrage fehl, erhält der Subscriber eine Fehlerantwort. Zum Beispiel:

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

Protokollmeldungen

Die Logs API ermöglicht es Erweiterungen, drei verschiedene Protokollstreams zu abonnieren:

- Funktion — Protokolliert, was die Lambda-Ereignisfunktion generiert und in `stdout` oder `stderr` schreibt.
- Erweiterung – Protokolle, die der Erweiterungscode generiert.
- Plattform – Protokolle, die von der Laufzeitplattform generiert werden und die Ereignisse und Fehler im Zusammenhang mit Aufrufen und Erweiterungen aufzeichnen.

Themen

- Funktionsprotokolle
- Erweiterungsprotokolle
- Plattformprotokolle

Funktionsprotokolle

Die Lambda Event Function und interne Erweiterungen generieren Funktionsprotokolle und schreiben sie in `stdout` oder`stderr`.

Das folgende Beispiel zeigt das Format einer Funktionslognachricht. {„time“: „2020-08-20“, T12:31:32.123Z „type“: „function“, „record“: „FEHLER aufgetreten. Stack trace:\n\my-function (line 10)\n" }

Erweiterungsprotokolle

Erweiterungen können Erweiterungsprotokolle generieren. Das Protokollformat ist das gleiche wie bei einem Funktionsprotokoll.

Plattformprotokolle

Lambda generiert Protokollmeldungen für Plattformereignisse wie `platform.start`, `platform.end` und `platform.fault`.

Optional können Sie die Version 2021-03-18 des Logs API-Schemas abonnieren, welche die `platform.runtimeDone`-Protokollmeldungen enthält.

Beispiel für Plattformprotokollmeldungen

Das folgende Beispiel zeigt die Plattformstart- und Plattformendprotokolle. Diese Protokolle zeigen die Startzeit des Aufrufs und die Endzeit des Aufrufs für den Aufruf an, den die RequestID angibt.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
```

```
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

Die RuntimeDoneplatform.init-Protokollnachricht zeigt den Status der `Runtime init` Unterphase, die Teil der Init-Lifecyle-Phase ist. Wenn `Runtime init` erfolgreich ist, sendet die Laufzeit eine `/next`-Laufzeit-API-Anfrage (für die `on-demand`- und `provisioned-concurrency`-Initialisierungstypen) oder `restore/next` (für den `snap-start`-Initialisierungstyp). Das folgende Beispiel zeigt eine erfolgreiche RuntimeDoneplatform.init-Protokollnachricht für den Initialisierungstyp. `snap-start`

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

Die platform.initReport-Protokollnachricht zeigt, wie lange die `Init`-Phase gedauert hat und wie viele Millisekunden Ihnen während dieser Phase in Rechnung gestellt wurden. Wenn der Initialisierungstyp `provisioned-concurrency` lautet, sendet Lambda diese Nachricht während des Aufrufs. Wenn der Initialisierungstyp `snap-start` lautet, sendet Lambda diese Nachricht nach der Wiederherstellung des Snapshots. Das folgende Beispiel zeigt eine platform.initReport-Protokollnachricht für den `snap-start`-Initialisierungstyp.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

Das Berichtsprotokoll der Plattform enthält Metriken über den Aufruf, den die RequestID angibt. Das `initDurationMs`-Feld ist nur dann im Protokoll enthalten, wenn der Aufruf einen Kaltstart

enthielt. Wenn die AWS X-Ray Ablaufverfolgung aktiv ist, enthält das Protokoll Metadaten. X-Ray Das folgende Beispiel zeigt ein Plattformberichtsprotokoll für einen Aufruf, der einen Kaltstart enthielt.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

Das Plattformfehlerprotokoll erfasst Fehler bei der Laufzeit oder der Ausführungsumgebung. Das folgende Beispiel zeigt eine Fehlerprotokollmeldung der Plattform.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können Sie geringfügige Unterschiede zwischen der Struktur und dem Inhalt von Systemprotokollnachrichten und Trace-Segmenten feststellen, die von verschiedenen Lambda-Ereignisfunktionen in Ihrem AWS-Konto ausgegeben werden.
> Eine der von dieser Änderung betroffenen Protokollausgaben ist das Feld `"record"` des Plattformfehlerprotokolls. Die folgenden Beispiele zeigen illustrative `"record"`-Felder im alten und neuen Format. Der neue Stil des Fehlerprotokolls enthält eine prägnantere Meldung Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Example Aufzeichnung des Plattformfehlerprotokolls (alter Stil)

```
"record":"RequestId: ...\tError: Runtime exited with error: exit status
 255\nRuntime.ExitError"
```

Example Aufzeichnung des Plattformfehlerprotokolls (neuer Stil)

```
"record":"RequestId: ... Status: error\tErrorType: Runtime.ExitError"
```

Lambda generiert ein Plattformerweiterungsprotokoll, wenn sich eine Erweiterung bei der Erweiterungs-API registriert. Das folgende Beispiel zeigt eine Plattformerweiterungsmeldung.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

Lambda generiert ein Plattformprotokoll-Abonnementprotokoll, wenn eine Erweiterung die Protokoll-API abonniert. Das folgende Beispiel zeigt eine Protokoll-Abonnementmeldung.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

Lambda generiert ein vom Plattformprotokoll abgelegtes Protokoll, wenn eine Erweiterung nicht in der Lage ist, die Anzahl der Protokolle zu verarbeiten, die sie empfängt. Das folgende Beispiel zeigt eine `platform.logsDropped`-Protokollmeldung.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
```

```
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
        "droppedBytes" 12345
    }
}
```

Die platform.restoreStart-Protokollnachricht zeigt den Zeitpunkt an, zu dem die `Restore`-Phase begonnen hat (nur `snap-start`-Initialisierungstyp). Beispiel:

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

Die platform.restoreReport-Protokollnachricht zeigt, wie lange die `Restore`-Phase gedauert hat und wie viele Millisekunden Ihnen während dieser Phase in Rechnung gestellt wurden (nur `snap-start`-Initialisierungstyp). Beispiel:

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

`RuntimeDone-Meldungen der Plattform`

Wenn Sie die Schemaversion in der Abonnementanforderung auf „2021-03-18" setzen, sendet Lambda eine `platform.runtimeDone` Nachricht, nachdem der Aufruf der Ereignisfunktion entweder erfolgreich oder mit einem Fehler abgeschlossen wurde. Die Erweiterung kann diese Nachricht verwenden, um die gesamte Telemetrieerfassung für diesen Aufruf der Event-Funktion zu beenden.

Die OpenAPI-Spezifikation für den Protokollereignistyp in der Schemaversion 2021-03-18 ist hier verfügbar: schema-2021-03-18.zip

Lambda generiert die `platform.runtimeDone`-Protokollmeldung, wenn die Laufzeitumgebung eine `Next`- oder `Error`-Laufzeit-API-Anforderung sendet. Das `platform.runtimeDone`-Protokoll informiert die Verbraucher über die Protokoll-API, dass der Funktionsaufruf abgeschlossen ist. Erweiterungen können diese Informationen verwenden, um zu entscheiden, wann die gesamte während dieses Aufrufs gesammelte Telemetrie gesendet werden soll.

Beispiele

Lambda sendet die `platform.runtimeDone`-Meldung, nachdem die Laufzeit die NEXT-Anfrage sendet, wenn der Funktionsaufruf abgeschlossen wird. Die folgenden Beispiele zeigen Meldungen für jeden der Statuswerte: Erfolg, Misserfolg und Timeout.

Example Beispiel Erfolgsmeldung

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example Beispiel Misserfolgsmeldung

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example Beispiel Timeout-Meldung

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
```

```
      "status": "timeout"
  }
}
```

Example Beispiel für eine RuntimeDone Platform.restore-Meldung (nur beim Initialisierungstyp **`Snapstart`**)

Die RuntimeDone Protokollnachricht platform.restore zeigt, ob die Phase erfolgreich war oder nicht. `Restore` Lambda sendet diese Nachricht, wenn die Laufzeit eine `restore/next`-Laufzeit-API-Anfrage sendet. Es gibt drei mögliche Status: erfolgreich, fehlgeschlagen und Timeout. Das folgende Beispiel zeigt eine erfolgreiche RuntimeDoneplatform.restore-Protokollnachricht.

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

Beheben von Problemen in Lambda

Die folgenden Themen enthalten Ratschläge zur Fehlerbehebung bei Fehlern und Problemen, die bei der Verwendung der Lambda-API, -Konsole oder -Tools auftreten können. Wenn Sie auf ein Problem stoßen, das hier nicht aufgeführt ist, können Sie die Schaltfläche Feedback auf dieser Seite verwenden, um es zu melden.

Weitere Tipps zur Fehlerbehebung und Antworten auf häufig gestellte Supportfragen finden Sie im AWS - Wissenscenter.

Weitere Informationen zum Debuggen und zur Problembehandlung für Lambda-Anwendungen finden Sie bei Serverless Land unter Debugging.

Topics

- Beheben von Konfigurationsproblemen in Lambda
- Beheben von Bereitstellungsproblemen in Lambda
- Beheben von Aufruf-Problemen in Lambda
- Beheben von Ausführungsproblemen in Lambda

- Behebung von Problemen mit der Zuordnung von Ereignisquellen in Lambda
- Beheben von Netzwerkproblemen in Lambda

Beheben von Konfigurationsproblemen in Lambda

Ihre Einstellungen für die Funktionskonfiguration können sich auf die Gesamtleistung und das Verhalten Ihrer Lambda-Ereignisfunktion auswirken. Diese führen möglicherweise nicht zu tatsächlichen Funktionsfehlern, können jedoch zu unerwarteten Timeouts und Ergebnissen führen.

Die folgenden Themen enthalten Tipps zur Fehlerbehebung für häufig auftretende Probleme im Zusammenhang mit den Konfigurationseinstellungen von Lambda-Funktionen.

Themen

- Arbeitsspeicherkonfigurationen
- CPU-bound Konfigurationen
- Timeouts
- Speicherlecks zwischen Aufrufen
- Asynchrone Ergebnisse wurden bei einem späteren Aufruf zurückgegeben

Arbeitsspeicherkonfigurationen

Sie können eine Lambda-Funktion so konfigurieren, dass sie zwischen 128 MB und 10 240 MB Speicher verwendet. Standardmäßig wird jeder in der Konsole erstellten Funktion die kleinste Speichermenge zugewiesen. Viele Lambda-Funktionen sind bei dieser niedrigsten Einstellung leistungsfähig. Wenn Sie jedoch große Codebibliotheken importieren oder speicherintensive Aufgaben ausführen, reichen 128 MB nicht aus.

Wenn Ihre Funktionen viel langsamer als erwartet ausgeführt werden, besteht der erste Schritt darin, die Speichereinstellung zu erhöhen. Bei speichergebundenen Funktionen behebt dies den Engpass und kann die Leistung Ihrer Funktion verbessern.

CPU-bound Konfigurationen

Wenn Ihre Funktion bei rechenintensiven Vorgängen langsamer als erwartet ausgeführt wird, kann dies daran liegen, dass Ihre Funktion CPU-bound In diesem Fall kann die Berechnungskapazität der Funktion nicht mit der Arbeit Schritt halten.

Lambda erlaubt Ihnen zwar nicht, die CPU-Konfiguration direkt zu ändern, aber die CPU wird indirekt über die Speichereinstellungen gesteuert. Der Lambda-Dienst weist proportional mehr virtuelle CPU zu, wenn Sie mehr Arbeitsspeicher zuweisen. Bei 1,8 GB Arbeitsspeicher wird einer Lambda-Funktion eine ganze vCPU zugewiesen und oberhalb dieser Grenze hat sie Zugriff auf mehr als einen vCPU-Kern. Mit 10 240 MB hat sie 6 vCPUs zur Verfügung. Mit anderen Worten, Sie können die Leistung verbessern, indem Sie die Speicherzuweisung erhöhen, auch wenn die Funktion nicht den gesamten Speicher verwendet.

Timeouts

Timeouts für Lambda-Funktionen können zwischen 1 und 900 Sekunden (15 Minuten) eingestellt werden. Für Funktionen, die AWS Lambda Managed Instances verwenden, unterstützen asynchrone Aufrufe und Aufrufe zur Zuordnung von Ereignisquellen (außer Amazon MQ und Amazon DocumentDB) einen zulässigen Höchstwert von 5.400 Sekunden (90 Minuten). Standardmäßig ist dieser Wert in der Lambda-Konsole auf 3 Sekunden eingestellt. Der Timeout-Wert ist eine Sicherheitsvorrichtung, die sicherstellt, dass Funktionen nicht unbegrenzt ausgeführt werden. Nachdem der Timeout-Wert erreicht ist, stoppt Lambda den Funktionsaufruf.

Wenn ein Timeout-Wert festgelegt wird, der nahe an der durchschnittlichen Dauer einer Funktion liegt, erhöht dies das Risiko, dass bei der Funktion ein unerwarteter Timeout auftritt. Die Dauer einer Funktion kann je nach Umfang der Datenübertragung und -verarbeitung sowie der Latenz aller Dienste, mit denen die Funktion interagiert, variieren. Häufige Ursachen für Timeouts:

- Beim Herunterladen von Daten aus S3-Buckets oder anderen Datenspeichern ist der Download größer oder dauert länger als im Durchschnitt.
- Eine Funktion sendet eine Anfrage an einen anderen Service, dessen Beantwortung länger dauert.
- Die einer Funktion zur Verfügung gestellten Parameter erfordern eine höhere Rechenkomplexität in der Funktion, wodurch der Aufruf länger dauert.

Stellen Sie beim Testen Ihrer Anwendung sicher, dass Ihre Tests die Größe und Menge der Daten sowie realistische Parameterwerte genau wiedergeben. Wichtig ist, dass Sie Datensätze verwenden, die an der Obergrenze dessen liegen, was für Ihren Workload vernünftigerweise zu erwarten ist.

Implementieren Sie zusätzlich Obergrenzen für Ihren Workload, wo immer dies möglich ist. In diesem Beispiel könnte die Anwendung für jeden Dateityp eine maximale Größenbeschränkung verwenden. Anschließend können Sie die Leistung Ihrer Anwendung für eine Reihe erwarteter Dateigrößen bis einschließlich der Höchstgrenzen testen.

Speicherlecks zwischen Aufrufen

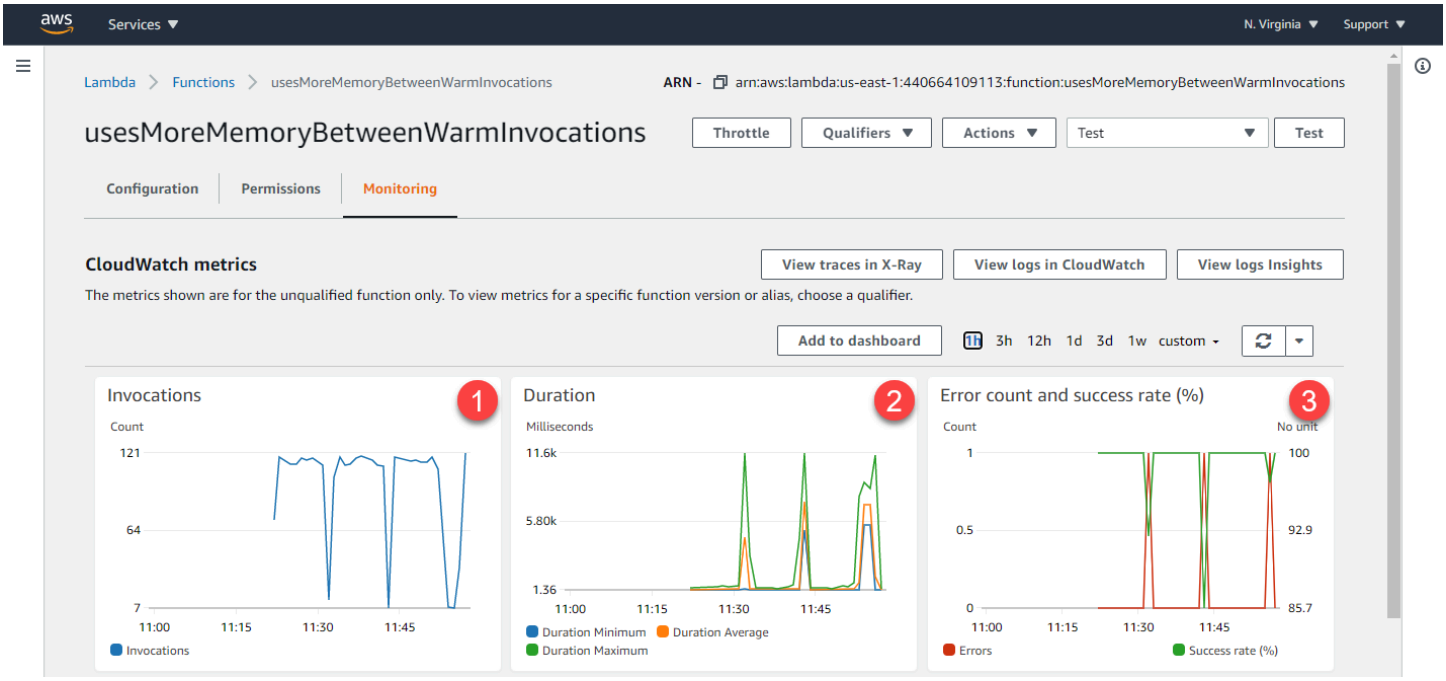
Globale Variablen und Objekte, die in der INIT-Phase eines Lambda-Aufrufs gespeichert sind, behalten ihren Status zwischen warmen Aufrufen bei. Sie werden erst vollständig zurückgesetzt, wenn die Ausführungsumgebung zum ersten Mal ausgeführt wird (auch als „Kaltstart“ bezeichnet). Alle im Handler gespeicherten Variablen werden zerstört, wenn der Handler beendet wird. Es hat sich bewährt, die INIT-Phase zu verwenden, um Datenbankverbindungen einzurichten, Bibliotheken zu laden, Caches zu erstellen und unveränderliche Ressourcen zu laden.

Wenn Sie Bibliotheken von Drittanbietern für mehrere Aufrufe in derselben Ausführungsumgebung verwenden, sollten Sie deren Dokumentation zur Verwendung in einer Serverless-Computerumgebung überprüfen. Einige Datenbankverbindungs- und Protokollierungsbibliotheken speichern möglicherweise Zwischenaufrufergebnisse und andere Daten. Dies führt dazu, dass der Speicherverbrauch dieser Bibliotheken mit nachfolgenden Warmaufrufen zunimmt. Wenn dies der Fall ist, stellen Sie möglicherweise fest, dass der Lambda-Funktion nicht genügend Speicher zur Verfügung steht, auch wenn Ihr benutzerdefinierter Code Variablen korrekt entsorgt.

Dieses Problem betrifft Aufrufe, die in warmen Ausführungsumgebungen auftreten. Der folgende Code erzeugt beispielsweise ein Speicherleck zwischen Aufrufen. Die Lambda-Funktion verbraucht bei jedem Aufruf zusätzlichen Speicher, indem sie die Größe eines globalen Arrays erhöht:

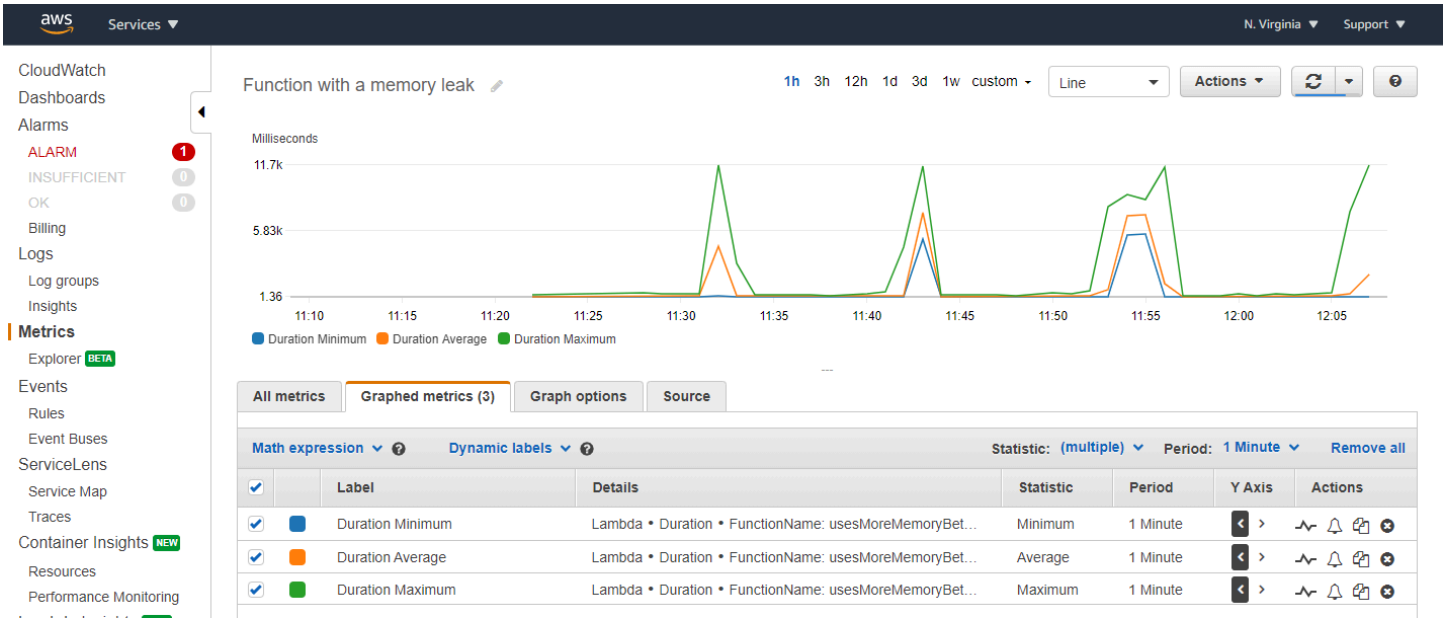
```
let a = []

exports.handler = async (event) => {
    a.push(Array(100000).fill(1))
}
```

Bei einer Konfiguration mit 128 MB Speicher zeigt die Registerkarte Überwachung der Lambda-Funktion nach 1 000 Aufrufen dieser Funktion die typischen Änderungen bei Aufrufen, Dauer und Fehleranzahl, wenn ein Speicherleck auftritt:



1. Aufrufe: Eine gleichmäßige Transaktionsrate wird periodisch unterbrochen, da die Aufrufe länger dauern, bis sie abgeschlossen sind. Im stationären Zustand verbraucht das Speicherleck nicht den gesamten der Funktion zugewiesenen Speicher. Wenn die Leistung abnimmt, paginiert das Betriebssystem den lokalen Speicher, um den wachsenden Speicherbedarf der Funktion zu decken, was dazu führt, dass weniger Transaktionen abgeschlossen werden.
2. Dauer: Bevor der Funktion der Speicherplatz ausgeht, beendet sie Aufrufe mit einer konstanten zweistelligen Millisekundenrate. Wenn ein Paging stattfindet, verlängert sich die Dauer um eine Größenordnung.
3. Anzahl der Fehler: Wenn der Speicherverlust den zugewiesenen Speicher übersteigt, kommt es schließlich zu einem Funktionsfehler, weil die Berechnung den Timeout überschreitet, oder die Ausführungsumgebung stoppt die Funktion.

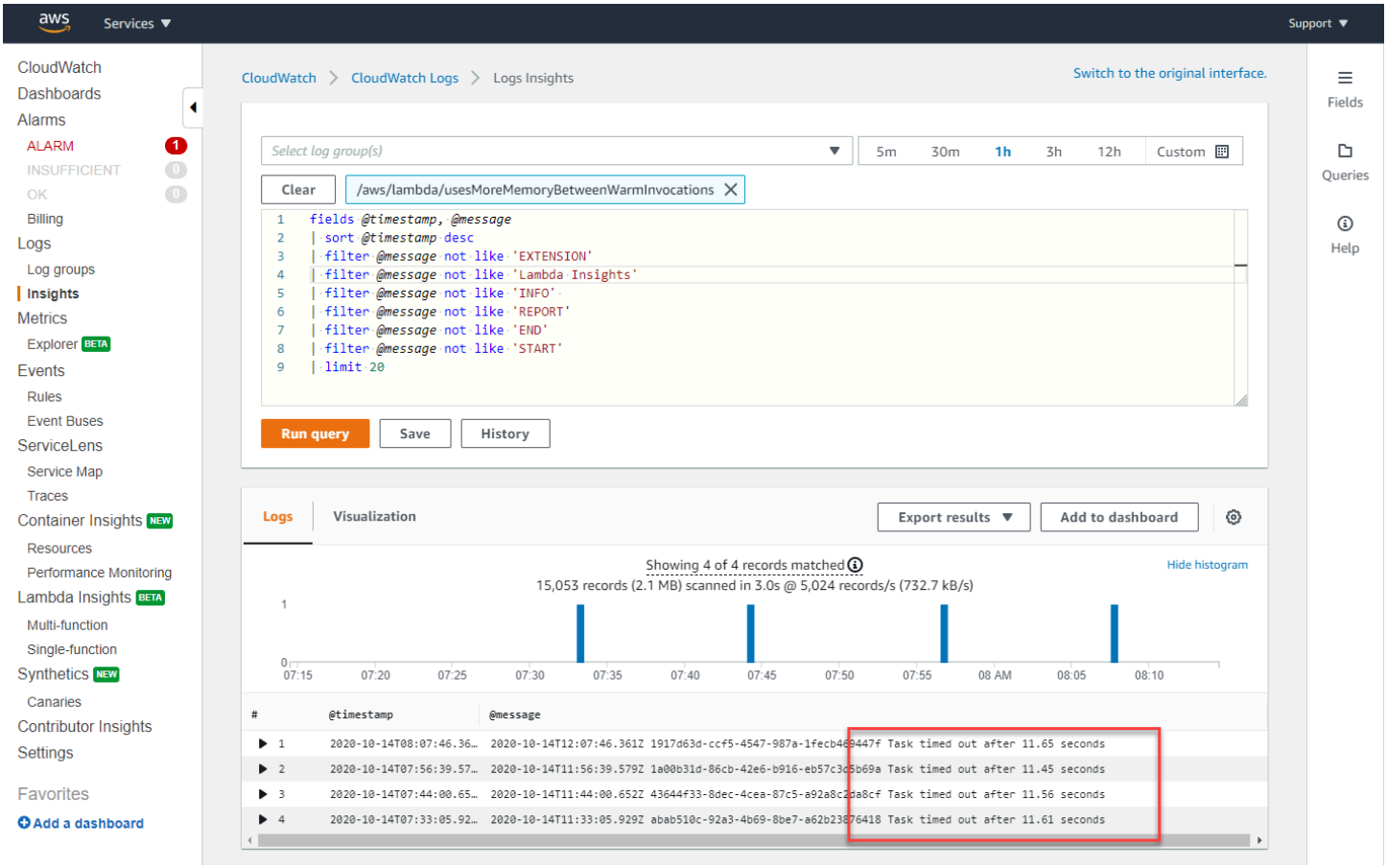
Nach dem Fehler startet Lambda die Ausführungsumgebung neu, was erklärt, warum alle drei Diagramme eine Rückkehr zum ursprünglichen Zustand zeigen. Wenn Sie die CloudWatch Messwerte für die Dauer erweitern, erhalten Sie detailliertere Angaben zu den Statistiken über die Mindest-, Höchst- und Durchschnittsdauer:



Um die Fehler zu finden, die bei den 1000 Aufrufen generiert wurden, können Sie die CloudWatch Insights-Abfragesprache verwenden. Die folgende Abfrage schließt Informationsprotokolle aus, um nur die Fehler zu melden:

```
fields @timestamp, @message
| sort @timestamp desc
| filter @message not like 'EXTENSION'
| filter @message not like 'Lambda Insights'
| filter @message not like 'INFO'
| filter @message not like 'REPORT'
| filter @message not like 'END'
| filter @message not like 'START'
```

Ein Abgleich mit der Protokollgruppe für diese Funktion zeigt, dass Zeitüberschreitungen für die periodischen Fehler verantwortlich waren:



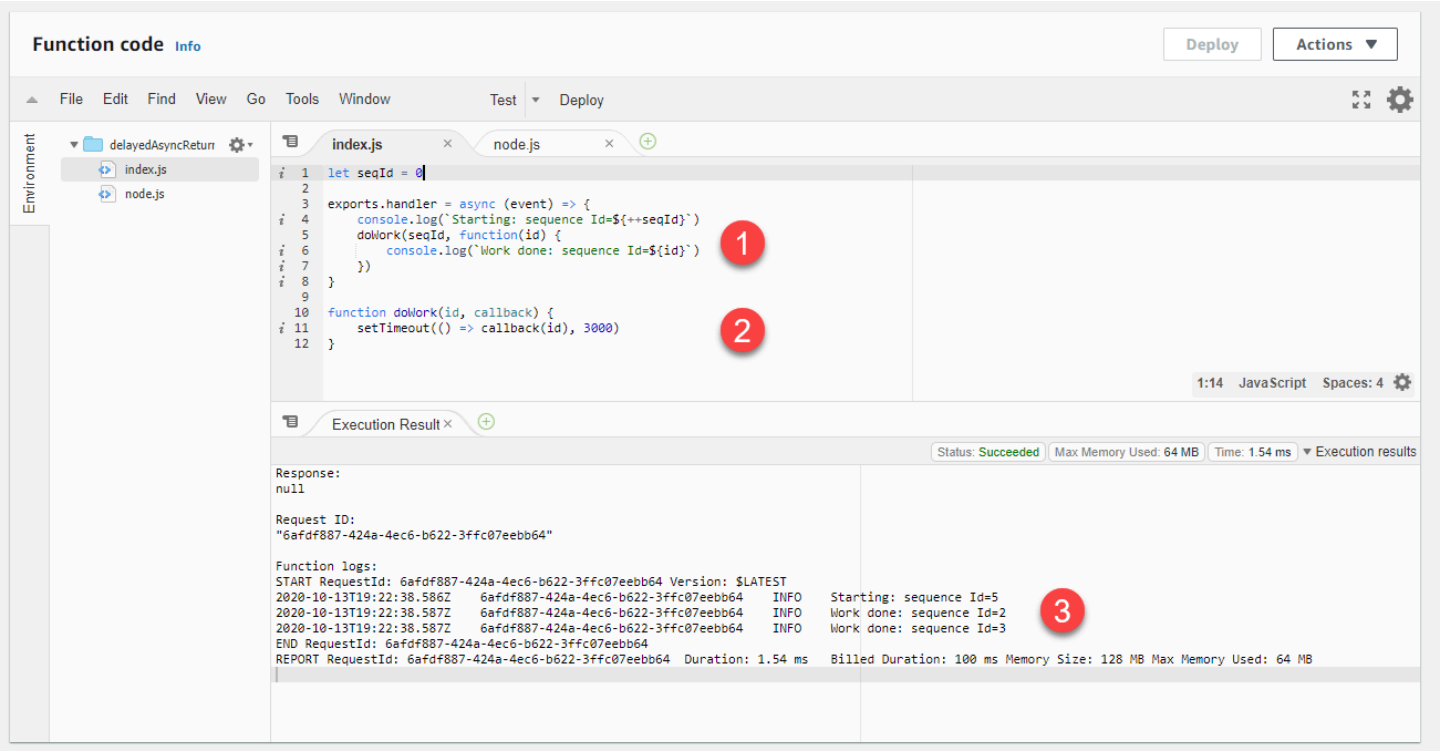
Asynchrone Ergebnisse wurden bei einem späteren Aufruf zurückgegeben

Bei Funktionscode, der asynchrone Muster verwendet, ist es möglich, dass die Callback-Ergebnisse eines Aufrufs bei einem future Aufruf zurückgegeben werden. In diesem Beispiel werden Node.js asynchrone Muster verwendet, aber dieselbe Logik kann auch auf andere Laufzeiten angewendet werden. Die Funktion verwendet die traditionelle Callback-Syntax in. JavaScript Sie ruft eine asynchrone Funktion mit einem inkrementellen Zähler auf, der die Anzahl der Aufrufe verfolgt:

```
let seqId = 0

exports.handler = async (event, context) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    doWork(seqId, function(id) {
        console.log(`Work done: sequence Id=${id}`)
    })
}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)
```

```
}
```

Wenn sie mehrmals hintereinander aufgerufen werden, treten die Ergebnisse der Callbacks bei nachfolgenden Aufrufen auf:



1. Der Code ruft die `doWork`-Funktion auf und stellt als letzten Parameter eine Callback-Funktion bereit.
2. Es dauert einige Zeit, bis die `doWork`-Funktion abgeschlossen ist, bevor der Callback aufgerufen wird.
3. Die Protokollierung der Funktion zeigt an, dass der Aufruf beendet wird, bevor die Ausführung der Funktion abgeschlossen ist. `doWork` Darüber hinaus werden nach dem Start einer Iteration Callbacks aus früheren Iterationen verarbeitet, wie in den Protokollen gezeigt.

In JavaScript werden asynchrone Callbacks mit einer Ereignisschleife behandelt. Andere Laufzeiten verwenden andere Mechanismen zur Behandlung der Gleichzeitigkeit. Wenn die Ausführungsumgebung der Funktion endet, friert Lambda die Umgebung bis zum nächsten Aufruf ein.

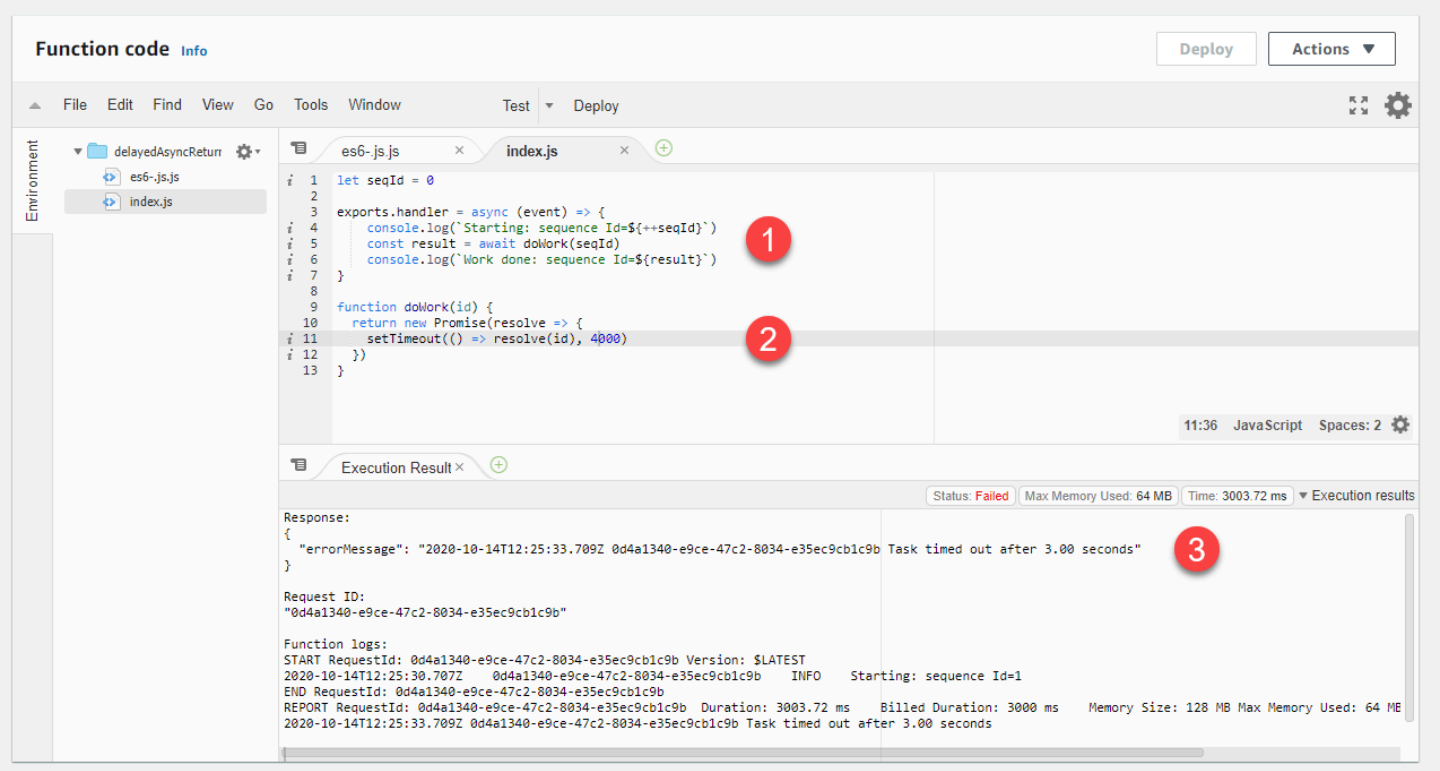
Nach der Wiederaufnahme wird die Verarbeitung der Ereignisschleife JavaScript fortgesetzt, die in diesem Fall einen asynchronen Rückruf von einem vorherigen Aufruf beinhaltet. Ohne diesen Kontext kann es den Anschein haben, dass die Funktion ohne Grund Code ausführt und beliebige Daten

zurückgibt. Sie ist vielmehr ein Artefakt der Interaktion zwischen der Gleichzeitigkeit zur Laufzeit und den Ausführungsumgebungen.

Dadurch besteht die Möglichkeit, dass private Daten aus einem früheren Aufruf in einem späteren Aufruf erscheinen. Es gibt zwei Möglichkeiten, die Funktion Ihres Handlers aufzuweisen. JavaScript Stellt zunächst die Schlüsselwörter async und await bereit, um die asynchrone Entwicklung zu vereinfachen und die Codeausführung zu erzwingen, auf den Abschluss eines asynchronen Aufrufs zu warten. Die obige Funktion kann mit diesem Ansatz wie folgt umgeschrieben werden:

```
let seqId = 0
exports.handler = async (event) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    const result = await doWork(seqId)
    console.log(`Work done: sequence Id=${result}`)
}

function doWork(id) {
  return new Promise(resolve => {
    setTimeout(() => resolve(id), 4000)
  })
}
```

Die Verwendung dieser Syntax verhindert, dass der Handler beendet wird, bevor die asynchrone Funktion beendet ist. Wenn in diesem Fall der Callback länger dauert als das Timeout der Lambda-Funktion, gibt die Funktion einen Fehler aus, anstatt das Callback-Ergebnis bei einem späteren Aufruf zurückzugeben:

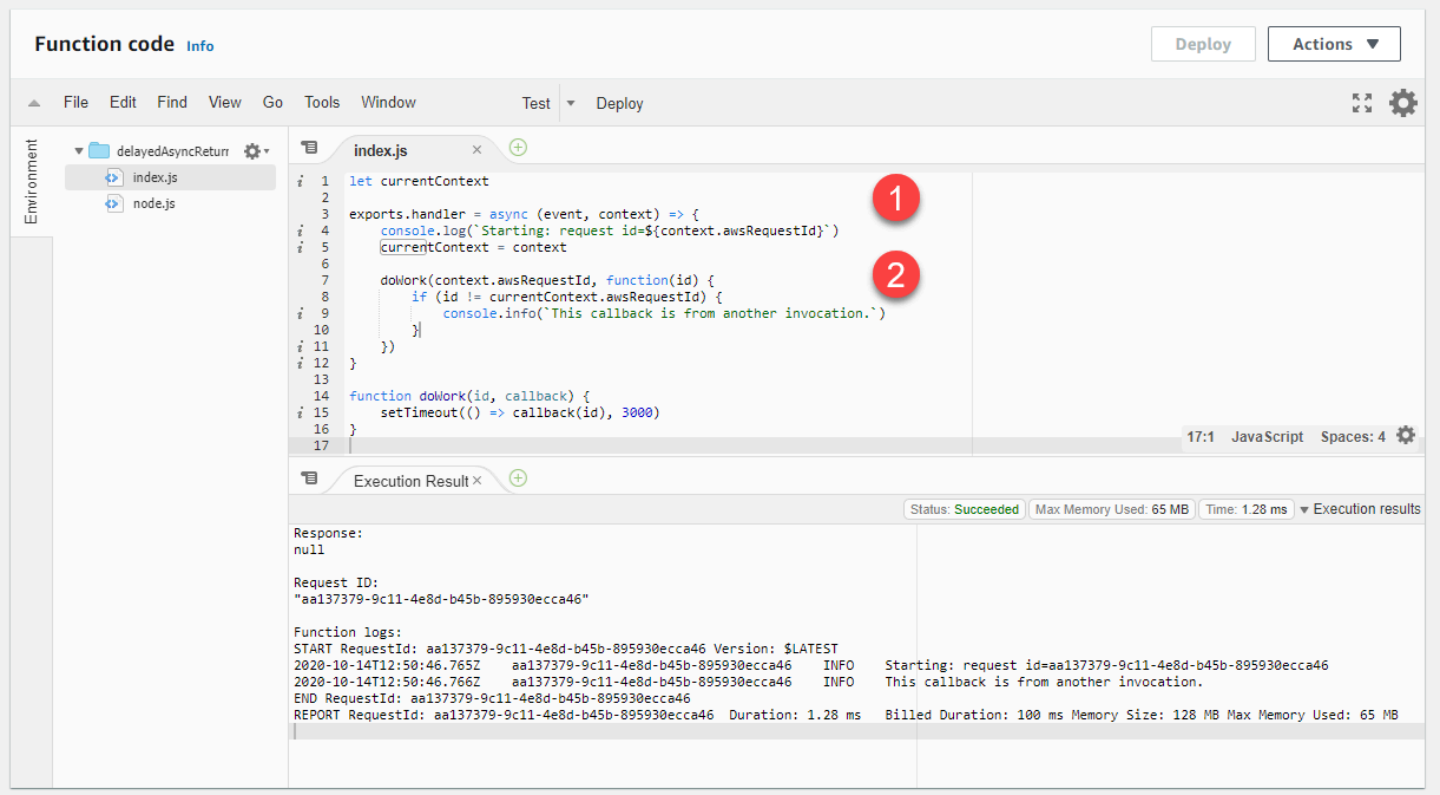


1. Der Code ruft die asynchrone `doWork`-Funktion unter Verwendung des Schlüsselworts „await“ im Handler auf.
2. Die Funktion `doWork` benötigt eine gewisse Zeit, bevor sie das Versprechen auflöst.
3. Bei der Funktion kommt es zu einem Timeout, da `doWork` länger dauert, als der Timeout erlaubt, und das Ergebnis des Rückrufs bei einem späteren Aufruf nicht zurückgegeben wird.

Im Allgemeinen sollten Sie sicherstellen, dass alle Hintergrundprozesse oder Rückrufe im Code abgeschlossen sind, bevor der Code beendet wird. Wenn dies in Ihrem Anwendungsfall nicht möglich ist, können Sie einen Bezeichner verwenden, um sicherzustellen, dass der Rückruf zum aktuellen Aufruf gehört. Dazu können Sie die vom Kontextobjekt RequestId bereitgestellte AWS verwenden. Indem Sie diesen Wert an den asynchronen Callback übergeben, können Sie den übergebenen Wert mit dem aktuellen Wert vergleichen, um festzustellen, ob der Callback von einem anderen Aufruf stammt:

```
let currentContext

exports.handler = async (event, context) => {
    console.log(`Starting: request id=$\{context.awsRequestId}`)
    currentContext = context
```

```
    doWork(context.awsRequestId, function(id) {
        if (id != currentContext.awsRequestId) {
            console.info(`This callback is from another invocation.`)
        }
    })

}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)

}
```



1. Der Lambda-Funktionshandler verwendet den Kontextparameter, der Zugriff auf eine eindeutige Aufrufanforderungs-ID ermöglicht.
2. Die `awsRequestId` wird an die doWork-Funktion übergeben. Im Callback wird die ID mit der `awsRequestId` des aktuellen Aufrufs verglichen. Wenn diese Werte unterschiedlich sind, kann der Code entsprechende Maßnahmen ergreifen.

Beheben von Bereitstellungsproblemen in Lambda

Wenn Sie Ihre Lambda-Ereignisfunktion aktualisieren, stellt Lambda die Änderung bereit, indem es neue Instanzen der Funktion mit dem aktualisierten Code oder den aktualisierten Einstellungen startet. Bereitstellungsfehler verhindern die Verwendung der neuen Version und können auf Probleme mit Bereitstellungspaket, Code, Berechtigungen oder Tools zurückzuführen sein.

Wenn Sie Updates für Ihre Funktion direkt mit der Lambda-API oder mit einem Client wie dem bereitstellen AWS CLI, können Sie Fehler von Lambda direkt in der Ausgabe sehen. Wenn Sie Dienste wie AWS CloudFormation, oder verwenden AWS CodeDeploy AWS CodePipeline, suchen Sie in den Protokollen oder im Eventstream für diesen Dienst nach der Antwort von Lambda.

Die folgenden Themen enthalten Ratschläge zur Fehlerbehebung bei Fehlern und Problemen, die bei der Verwendung der Lambda-API, -Konsole oder -Tools auftreten können. Wenn Sie auf ein Problem stoßen, das hier nicht aufgeführt ist, können Sie die Schaltfläche Feedback auf dieser Seite verwenden, um es zu melden.

Weitere Tipps zur Fehlerbehebung und Antworten auf häufig gestellte Supportfragen finden Sie im AWS - Wissenscenter.

Weitere Informationen zum Debuggen und zur Problembehandlung für Lambda-Anwendungen finden Sie bei Serverless Land unter Debugging.

Themen

- Allgemein: Der Zugriff wurde verweigert oder die Datei kann nicht geladen werden
- Allgemein: Beim Aufrufen von tritt ein Fehler auf UpdateFunctionCode
- Amazon S3: Fehlercode PermanentRedirect.
- Allgemein: Kann nicht gefunden werden, kann nicht geladen werden, Klasse nicht gefunden, keine solche Datei oder kein Verzeichnis
- Allgemein: Undefinierter Methodenhandler
- Allgemein: Speicherlimit für Lambda-Code überschritten
- Lambda: Ebenenkonvertierung ist fehlgeschlagen.
- Lambda: oder InvalidParameterValueException RequestEntityTooLargeException
- Lambda: InvalidParameterValueException
- Lambda: Kontingente für Gleichzeitigkeit und Speicher

- Lambda: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit

Allgemein: Der Zugriff wurde verweigert oder die Datei kann nicht geladen werden

Fehler: EACCES: Zugriff verweigert, öffnen Sie '/ /index.js 'var/task

Fehler: kann eine derartige Datei nicht laden – Funktion

Fehler: [Errno 13] Zugriff verweigert: '/ /function.py 'var/task

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Allgemein: Beim Aufrufen von tritt ein Fehler auf UpdateFunctionCode

Fehler: Beim Aufrufen der UpdateFunctionCode Operation ist ein Fehler aufgetreten (RequestEntityTooLargeException)

Wenn Sie ein Bereitstellungspaket oder ein Ebenenarchiv direkt in Lambda hochladen, ist die Größe der ZIP-Datei auf 50 MB begrenzt. Um eine größere Datei hochzuladen, speichern Sie sie in Amazon S3 und verwenden Sie die S3Bucket- und S3Key--Parameter.

> **Note**
>
> Wenn Sie eine Datei direkt mit dem AWS CLI AWS SDK oder auf andere Weise hochladen, wird die binäre ZIP-Datei in ein Base64-Format konvertiert, wodurch sich ihre Größe um etwa 30% erhöht. Um dies und die Größe anderer Parameter in der Anforderung zu ermöglichen, ist die tatsächliche von Lambda angewendete Größenbeschränkung der Anforderung größer. Aus diesem Grund ist der Grenzwert von 50 MB ein ungefährer Wert.

Amazon S3: Fehlercode PermanentRedirect.

Fehler: Während ist ein Fehler aufgetreten GetObject. S3-Fehlercode: PermanentRedirect. S3-Fehlermeldung: Der Bucket befindet sich in dieser Region: us-east-2. Verwenden Sie diese Region, um die Anfrage zu wiederholen

Wenn Sie das Bereitstellungspaket einer Funktion aus einem Amazon-S3-Bucket hochladen, muss sich der Bucket in derselben Region wie die Funktion befinden. Dieses Problem kann auftreten, wenn Sie ein Amazon S3 S3-Objekt in einem Aufruf angeben oder die Befehle package und deploy in der AWS CLI oder AWS SAM CLI verwenden. UpdateFunctionCode Erstellen Sie einen Bucket mit Bereitstellungsartefakt für jede Region, in der Sie Anwendungen entwickeln.

Allgemein: Kann nicht gefunden werden, kann nicht geladen werden, Klasse nicht gefunden, keine solche Datei oder kein Verzeichnis

Fehler: Modul 'function' kann nicht gefunden werden

Fehler: kann eine derartige Datei nicht laden – Funktion

Fehler: Modul 'function' kann nicht importiert werden

Fehler: Klasse nicht gefunden: function.Handler

Fehler: fork/exec /var/task/function: keine solche Datei oder kein solches Verzeichnis

Fehler: Der Typ '' konnte nicht aus der Assembly Function.Handler 'Function' geladen werden.

Der Name der Datei oder Klasse in der Handler-Konfiguration Ihrer Funktion stimmt nicht mit Ihrem Code überein. Weitere Informationen finden Sie im folgenden Abschnitt .

Allgemein: Undefinierter Methodenhandler

Fehler: index.handler ist nicht definiert oder wurde nicht exportiert

Fehler: Handler 'handler' fehlt für Modul 'function'

Fehler: undefinierte Methode `Handler' für #< > LambdaHandler:0x000055b76ccebf98

Fehler: Keine öffentliche Methode namens handleRequest mit der entsprechenden Methodensignatur für Klasse function.Handler gefunden

Fehler: Die Methode 'handleRequest' im Typ '' aus der Assembly 'FunctionFunction.Handler' konnte nicht gefunden werden

Der Name der Handler-Methode in der Handler-Konfiguration Ihrer Funktion stimmt nicht mit Ihrem Code überein. Jede Laufzeit definiert eine Namenskonvention für Handler, wie z. *`filename methodname`*. Der Handler ist die Methode im Code Ihrer Funktion, die von der Laufzeit ausführt wird, wenn Ihre Funktion aufgerufen wird.

Lambda stellt für einige Sprachen eine Bibliothek mit einer Schnittstelle bereit, die erwartet, dass eine Handler-Methode einen bestimmten Namen hat. Weitere Informationen zur Handlerbenennung für jede Sprache finden Sie in den folgenden Themen.

- Erstellen von Lambda-Eventfunktionen mit Node.js
- Lambda-Ereignisfunktionen mit Python erstellen
- Lambda-Ereignisfunktionen mit Ruby erstellen
- Lambda-Ereignisfunktionen mit Java erstellen
- Erstellen von Lambda-Eventfunktionen mit Go
- Lambda-Ereignisfunktionen mit C# erstellen
- Lambda-Ereignisfunktionen erstellen mit PowerShell

Allgemein: Speicherlimit für Lambda-Code überschritten

Fehler: Das Code-Speicherlimit wurde überschritten.

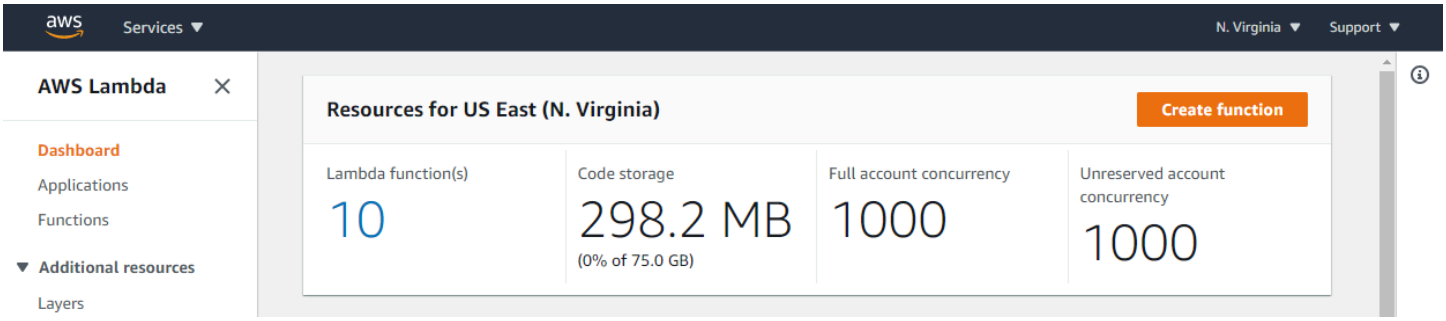
Lambda speichert eine Kopie Ihres Funktionscodes in einem internen S3-Bucket, der für Ihr Konto privat ist. Jedem AWS Konto werden in jeder Region 300 GB Codespeicher (entpackt) zugewiesen. Der Codespeicher umfasst den gesamten Speicherplatz, der sowohl von Lambda-Funktionen als auch von Lambda-Ebenen verwendet wird. Wenn Sie das Kontingent erreichen, erhalten Sie eine, CodeStorageExceededExceptionwenn Sie versuchen, neue Funktionen bereitzustellen.

Um dieses Problem zu beheben, können Sie alte Versionen von Funktionen bereinigen, ungenutzten Code entfernen oder Lambda-Ebenen verwenden, um gemeinsamen Code zu deduplizieren, der von

Funktionen gemeinsam genutzt wird. Darüber hinaus empfiehlt es sich, separate AWS Konten für separate Workloads zu verwenden, um die Verwaltung von Speicherkontingenten zu erleichtern.

Alternativ können Sie Ihre Funktionen und Ebenen so konfigurieren, dass sie den selbstverwalteten S3-Codespeicher verwenden. Mit dem selbstverwalteten S3-Codespeicher referenziert Lambda Ihren Code direkt aus Ihrem S3-Bucket, was nicht auf Ihr Speicherkontingent angerechnet wird.

Sie können Ihre gesamte Speichernutzung in der Lambda-Konsole im Untermenü Dashboard einsehen:



Lambda: Ebenenkonvertierung ist fehlgeschlagen.

Fehler: Lambda-Ebenenkonvertierung ist fehlgeschlagen. Hinweise zur Lösung dieses Problems finden Sie auf der Seite „Beheben von Bereitstellungsproblemen in Lambda“ im Lambda-Benutzerhandbuch.

Wenn Sie eine Lambda-Funktion mit einer Ebene konfigurieren, führt Lambda die Ebene mit Ihrem Funktionscode zusammen. Wenn dieser Prozess nicht abgeschlossen wird, gibt Lambda diesen Fehler zurück. Führen Sie in diesem Fall die folgenden Schritte aus:

- Löschen Sie alle ungenutzten Dateien von Ihrer Ebene.
- Löschen Sie alle symbolischen Links in Ihrer Ebene.
- Benennen Sie alle Dateien um, die denselben Namen wie ein Verzeichnis in einer der Ebenen Ihrer Funktion haben.

Lambda: oder InvalidParameterValueException RequestEntityTooLargeException

FehlerInvalidParameterValueException: Lambda konnte Ihre Umgebungsvariablen nicht konfigurieren, da die von Ihnen angegebenen Umgebungsvariablen das Limit von 4 KB überschritten haben. Gemessene Zeichenfolge: {"A1":" u... SFeY5cyPiPn7AtnX5BsM

Fehler:RequestEntityTooLargeException: Die Anfrage muss für den Vorgang kleiner als 5120 Byte sein UpdateFunctionConfiguration

Die maximale Größe des Variablenobjekts, das in der Konfiguration der Funktion gespeichert ist, darf 4096 Bytes nicht überschreiten. Dazu gehören Schlüsselnamen, Werte, Anführungszeichen, Kommas und Klammern. Die Gesamtgröße des HTTP-Anforderungstexts ist ebenfalls begrenzt.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    ...
}
```

In diesem Beispiel umfasst das Objekt 39 Zeichen und benötigt 39 Bytes, wenn es als Zeichenfolge gespeichert wird (ohne Leerzeichen `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}`. Standard-ASCII-Zeichen in Umgebungsvariablenwerten verwenden jeweils ein Byte. Erweiterte ASCII- und Unicode-Zeichen können zwischen 2 und 4 Bytes pro Zeichen verwenden.

Lambda: InvalidParameterValueException

FehlerInvalidParameterValueException: Lambda konnte Ihre Umgebungsvariablen nicht konfigurieren, da die von Ihnen angegebenen Umgebungsvariablen reservierte Schlüssel enthalten, deren Änderung derzeit nicht unterstützt wird.

Lambda reserviert einige Umgebungsvariablenschlüssel zur internen Verwendung. Beispielsweise wird `AWS_REGION` von der Laufzeit verwendet, um die aktuelle Region zu bestimmen, und kann nicht außer Kraft gesetzt werden. Andere Variablen, wie z. B. `PATH`, werden von der Laufzeit verwendet, können aber in Ihrer Funktionskonfiguration erweitert werden. Eine vollständige Liste finden Sie unter Definierte Laufzeitumgebungsvariablen.

Lambda: Kontingente für Gleichzeitigkeit und Speicher

Fehler: Die ConcurrentExecutions für die Funktion angegebene Funktion senkt den Wert des Kontos UnreservedConcurrentExecution unter den Mindestwert

Fehler: Der Wert MemorySize '' hat die Einschränkung nicht erfüllt: Das Mitglied muss einen Wert haben, der kleiner oder gleich 3008 ist

Diese Fehler treten auf, wenn Sie für Ihr Konto die Kontingente für Gleichzeitigkeit oder Speicher überschreiten. Bei neuen AWS Konten wurden Parallelität und Speicherkontingente reduziert. Um Fehler im Zusammenhang mit der Gleichzeitigkeit zu beheben, können Sie eine Kontingenterhöhung beantragen. Sie können keine Erhöhung des Speicherkontingents beantragen.

- Parallelität: Ein Fehler kann auftreten, wenn Sie wenn Sie versuchen, eine Funktion mit reservierter oder bereitgestellter Parallelität zu erstellen, oder wenn Ihre funktionsspezifische Parallelitätsanforderung (PutFunctionConcurrency) das Parallelitätskontingent Ihres Kontos übersteigt.
- Speicher: Fehler treten auf, wenn die der Funktion zugewiesene Speichermenge das Speicherkontingent Ihres Kontos übersteigt.

Lambda: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit

Fehler: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit

Dieser Fehler tritt auf, wenn Sie versuchen, den Code oder die Konfiguration einer Lambda-Funktion zu aktualisieren, während ein Alias mit bereitgestellter Gleichzeitigkeit auf eine Version verweist, bei der Probleme auftreten. Lambda initialisiert Ausführungsumgebungen für die bereitgestellte Gleichzeitigkeit vorab. Wenn diese Umgebungen aufgrund von Codefehlern, Ressourcenbeschränkungen oder einem betroffenen Stack und Alias nicht ordnungsgemäß initialisiert werden können, schlägt die Bereitstellung fehl. Führen Sie in diesem Fall die folgenden Schritte aus:

1. Den Alias rückgängig machen: Aktualisieren Sie den Alias vorübergehend, sodass er auf die zuvor funktionierende Version verweist.

```
aws lambda update-alias \
 --function-name <function-name> \
 --name <alias-name> \
 --function-version <known-good-version>
```

2. Lambda-Initialisierungscode korrigieren: Stellen Sie sicher, dass der Initialisierungscode, der außerhalb des Handlers ausgeführt wird, keine nicht abgefangenen Ausnahmen enthält, und initialisieren Sie die Clients und Verbindungen.
3. Sicherheit erneut bereitstellen: Stellen Sie korrigierten Code bereit und veröffentlichen Sie eine neue Version. Aktualisieren Sie dann den Alias so, dass er auf die korrigierte Version verweist. Aktivieren Sie optional die bereitgestellte Gleichzeitigkeit erneut, falls erforderlich.

Falls Sie dies verwenden AWS CloudFormation, aktualisieren Sie die Stack-Definition, `FunctionVersion:!GetAtt version.Version` sodass der Alias auf die funktionierende Version verweist:

```
alias:
 Type: AWS::Lambda::Alias
 Properties:
 FunctionName: !Ref function
FunctionVersion: !GetAtt version.Version
 Name: BLUE
 ProvisionedConcurrencyConfig:
 ProvisionedConcurrentExecutions: 1
```

Beheben von Aufruf-Problemen in Lambda

Wenn Sie eine Lambda-Ereignisfunktion aufrufen, validiert Lambda die Anforderung und prüft die Skalierungskapazität, bevor das Ereignis an Ihre Funktion oder, bei asynchronem Aufruf, an die Ereigniswarteschlange gesendet wird. Aufruffehler können durch Probleme mit Anforderungsparametern, Ereignisstruktur, Funktionseinstellungen, Benutzerberechtigungen, Ressourcenberechtigungen oder Grenzwerten verursacht werden.

Wenn Sie Ihre Funktion direkt aufrufen, sehen Sie Aufruffehler in der Antwort von Lambda. Wenn Sie Ihre Funktion asynchron mit einer Ereignisquellen-Zuweisung oder über einen anderen Service aufrufen, finden Sie möglicherweise Fehler in Protokollen, eine Warteschlange für unzustellbare Nachrichten oder ein Ziel für fehlerhafte Ereignisse. Die Optionen für die Fehlerbehandlung und das Wiederholungsverhalten variieren je nachdem, wie Sie Ihre Funktion aufrufen, und nach der Art des Fehlers.

Eine Liste der Fehlertypen, die der `Invoke`-Vorgang zurückgeben kann, finden Sie unter Aufrufen.

Themen

- Lambda: Bei der Funktion tritt während der Init-Phase () ein Timeout auf Sandbox.Timedout
- IAM: Lambda: nicht autorisiert InvokeFunction
- Lambda: Es konnte kein gültiges Bootstrap () gefunden werden Runtime.InvalidEntrypoint
- Lambda: Vorgang kann nicht ausgeführt werden ResourceConflictException
- Lambda: Funktion bleibt im Status Pending
- Lambda: Eine Funktion verwendet alle Parallelität
- Allgemein: Funktion kann nicht mit anderen Konten oder Diensten aufgerufen werden
- Allgemein: Funktionsaufruf bleibt im Looping
- Lambda: Alias-Routing mit bereitgestellter Parallelität
- Lambda: Kaltstarts mit bereitgestellter Parallelität
- Lambda: Kaltstarts mit neuen Versionen
- Lambda: Unerwarteter Node.js Abbruch in Runtime () Runtime.NodejsExit
- EFS: Die Funktion konnte das EFS-Dateisystem nicht mounten
- EFS: Die Funktion konnte keine Verbindung zum EFS-Dateisystem herstellen
- EFS: Die Funktion konnte das EFS-Dateisystem aufgrund eines Timeouts nicht mounten
- S3-Dateien: Die Funktion konnte das S3-Dateisystem nicht mounten
- S3-Dateien: Die Funktion konnte keine Verbindung zum S3-Dateisystem herstellen
- S3-Dateien: Die Funktion konnte das S3-Dateisystem aufgrund eines Timeouts nicht mounten
- Lambda: Lambda hat einen IO-Prozess erkannt, der zu lange dauerte
- Container CodeArtifactUserException : Fehler
- Container: InvalidEntrypoint Fehler

Lambda: Bei der Funktion tritt während der Init-Phase () ein Timeout auf Sandbox.Timedout

Fehler: Zeitüberschreitung der Aufgabe nach 3,00 Sekunden

Wenn die Init-Phase abläuft, initialisiert Lambda die Ausführungsumgebung erneut, indem es die `Init`-Phase erneut durchläuft, wenn die nächste Aufrufanforderung eintrifft. Dies wird als unterdrückte Initialisierung bezeichnet. Wenn Ihre Funktion jedoch mit einer kurzen Timeout-Dauer (in der Regel etwa 3 Sekunden) konfiguriert ist, kann es sein, dass die unterdrückte Initialisierung während der zugewiesenen Timeout-Dauer nicht abgeschlossen wird, sodass für die `Init`-Phase

erneut ein Timeout auftritt. Alternativ wird die unterdrückte Initialisierung abgeschlossen, es bleibt aber nicht genügend Zeit für den Abschluss der Invoke-Phase übrig, was zu einem Timeout für die `Invoke`-Phase führt.

Um Timeout-Fehler zu reduzieren, verwenden Sie eine oder mehrere der folgenden Strategien:

- Erhöhen Sie die Dauer des Funktions-Timeouts: Verlängern Sie das Timeout, um den `Init`- und `Invoke`-Phasen Zeit zu geben, erfolgreich abgeschlossen zu werden.
- Erhöhen Sie die Zuweisung des Funktionsspeichers: Mehr Arbeitsspeicher bedeutet auch eine proportionalere CPU-Zuweisung, wodurch sowohl die `Init`-Phase als auch die `Invoke`-Phasen beschleunigt werden können.
- Optimieren Sie den Code für die Funktionsinitialisierung: Reduzieren Sie die für die Initialisierung benötigte Zeit, um sicherzustellen, dass die `Init`- und `Invoke`-Phase innerhalb der konfigurierten Zeitspanne abgeschlossen werden kann.

IAM: Lambda: nicht autorisiert InvokeFunction

Fehler: Benutzer: arn:aws:iam: :123456789012: ist nicht berechtigt, Folgendes auszuführen: lambda: on resource: my-function user/developer InvokeFunction

Ihr Benutzer oder die Rolle, die Sie annehmen, muss über die Berechtigung zum Aufrufen einer Funktion verfügen. Diese Anforderung gilt auch für Lambda-Funktionen und andere Datenverarbeitungsressourcen, die Funktionen aufrufen. Fügen Sie Ihrem Benutzer die AWS verwaltete Richtlinie hinzu AWSLambdaRole, oder fügen Sie eine benutzerdefinierte Richtlinie hinzu, die die Aktion für die Zielfunktion ermöglicht. `lambda:InvokeFunction`

> **Note**
>
> Der Name der IAM-Aktion (`lambda:InvokeFunction`) bezieht sich auf den `Invoke`-Lambda-API-Vorgang.

Weitere Informationen finden Sie unter Verwaltung von Berechtigungen in AWS Lambda.

Lambda: Es konnte kein gültiges Bootstrap () gefunden werden Runtime.InvalidEntrypoint

Fehler: Es konnten keine gültigen Bootstrap (s) gefunden werden: [//bootstrap/var/task] opt/bootstrap

Dieser Fehler tritt normalerweise auf, wenn das Stammverzeichnis Ihres Bereitstellungspakets keine ausführbare Datei mit dem Namen `bootstrap` enthält. Wenn Sie beispielsweise eine `provided.al2023`-Funktion über eine ZIP-Datei bereitstellen, muss sich die `bootstrap`-Datei im Stammverzeichnis der ZIP-Datei befinden, nicht in einem Verzeichnis.

Lambda: Vorgang kann nicht ausgeführt werden ResourceConflictException

FehlerResourceConflictException: Der Vorgang kann derzeit nicht ausgeführt werden. Die Funktion befindet sich derzeit in folgendem Zustand: Ausstehend

Wenn Sie zum Zeitpunkt der Erstellung eine Funktion mit einer Virtual Private Cloud (VPC) verbinden, wechselt die Funktion in einen `Pending`-Zustand, während Lambda Elastic-Network-Schnittstellen erstellt. Während dieser Zeit können Sie Ihre Funktion nicht aufrufen oder ändern. Wenn Sie Ihre Funktion nach der Erstellung mit einer VPC verbinden, können Sie sie aufrufen, während das Update aussteht, aber Sie können den Code oder die Konfiguration nicht ändern.

Weitere Informationen finden Sie unter Status der Lambda-Ereignisfunktion.

Lambda: Funktion bleibt im Status Pending

Fehler: Eine Funktion bleibt mehrere Minuten im *`Pending`*-Zustand.

Wenn eine Funktion länger als sechs Minuten im `Pending`-Zustand ist, rufen Sie eine der folgenden API-Operationen auf, um die Blockierung aufzuheben.

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda bricht den ausstehenden Vorgang ab und versetzt die Funktion in den `Failed`-Zustand. Sie können dann versuchen, eine weitere Aktualisierung durchzuführen.

Lambda: Eine Funktion verwendet alle Parallelität

Problem: Eine Funktion verwendet die gesamte verfügbare Gleichzeitigkeit, wodurch andere Funktionen gedrosselt werden.

Um die verfügbare Parallelität Ihres AWS Kontos in einer AWS Region in Pools aufzuteilen, verwenden Sie reservierte Parallelität. Durch reservierte Gleichzeitigkeit wird sichergestellt, dass eine

Funktion immer auf ihre zugewiesene Gleichzeitigkeit skalieren kann und nicht über die zugewiesene Gleichzeitigkeit hinaus skaliert wird.

Allgemein: Funktion kann nicht mit anderen Konten oder Diensten aufgerufen werden

Problem: Sie können Ihre Funktion direkt aufrufen. Sie wird jedoch nicht ausgeführt, wenn sie von einem anderen Service oder Konto aufgerufen wird.

Sie erteilen anderen Services und Konten die Berechtigung zum Aufrufen einer Funktion in der ressourcenbasierten Richtlinieder Funktion. Wenn sich der Aufrufer in einem anderen Konto befindet, benötigt dieser Benutzer auch die Berechtigung zum Aufrufen von Funktionen.

Allgemein: Funktionsaufruf bleibt im Looping

Problem: Die Funktion wird kontinuierlich in einer Schleife aufgerufen.

Dies ist in der Regel der Fall, wenn Ihre Funktion Ressourcen in demselben AWS Dienst verwaltet, der sie auslöst. Beispielsweise ist es möglich, eine Funktion zu erstellen, die ein Objekt in einem Bucket von Amazon Simple Storage Service (Amazon S3) speichert, der mit einer Benachrichtigung konfiguriert ist, die die Funktion erneut aufruft. Um die Ausführung der Funktion zu stoppen, reduzieren Sie die verfügbare Gleichzeitigkeit auf Null, wodurch alle zukünftigen Aufrufe gedrosselt werden.

Identifizieren Sie dann den Codepfad oder Konfigurationsfehler, der den rekursiven Aufruf verursacht hat. Lambda erkennt und stoppt automatisch rekursive Schleifen für einige AWS Dienste und SDKs. Weitere Informationen finden Sie unter the section called “Erkennung rekursiver Schleifen”.

Lambda: Alias-Routing mit bereitgestellter Parallelität

Problem: Bereitgestellte Parallelitäts-Überlauf-Aufrufe während des Alias-Routings.

Lambda verwendet ein einfaches probabilistisches Modell, um den Datenverkehr zwischen den beiden Funktionsversionen zu verteilen. Bei niedrigem Datenverkehr sehen Sie möglicherweise eine hohe Abweichung zwischen dem konfigurierten und dem tatsächlichen Prozentsatz des Datenverkehrs für jede Version. Wenn Ihre Funktion bereitgestellte Parallelität verwendet, können Sie Überlaufaufrufe durch Konfigurieren einer höheren Anzahl von bereitgestellten Parallelitätsinstances während des aktiven Alias-Routings vermeiden.

Lambda: Kaltstarts mit bereitgestellter Parallelität

Problem: Sie erleben Kaltstarts nach dem Aktivieren der bereitgestellten Parallelität.

Wenn die Anzahl der gleichzeitigen Ausführungen einer Funktion kleiner oder gleich der konfigurierten Ebene der bereitgestellten Parallelität ist, sollte es keine Kaltstarts geben. Um Ihnen zu helfen zu bestätigen, ob die bereitgestellte Parallelität normal funktioniert, gehen Sie wie folgt vor:

- Überprüfen Sie, ob die bereitgestellte Parallelität auf der Funktionsversion oder dem Alias aktiviert ist.

  > **Note**
  >
  > Die bereitgestellte Gleichzeitigkeit kann in der unveröffentlichten Version der Funktion nicht konfiguriert werden.

- Stellen Sie sicher, dass Ihre Auslöser die richtige Funktionsversion oder den richtigen Alias aufrufen. Wenn Sie zum Beispiel Amazon API Gateway verwenden, überprüfen Sie, dass Amazon API die Funktionsversion oder den Alias mit bereitgestellter Parallelität aufruft, nicht $LATEST. Um zu überprüfen, ob die bereitgestellte Parallelität verwendet wird, können Sie die ProvisionedConcurrencyInvocations CloudWatch Amazon-Metrik überprüfen. Ein Wert ungleich Null gibt an, dass die Funktion Aufrufe in initialisierten Ausführungsumgebungen verarbeitet.
- Stellen Sie anhand der Metrik fest, ob Ihre Funktionsparallelität die konfigurierte Stufe der bereitgestellten Parallelität überschreitet. ProvisionedConcurrencySpilloverInvocations CloudWatch Ein Wert ungleich Null gibt an, dass die gesamte bereitgestellte Parallelität verwendet wird und einige Aufrufe mit einem Kaltstart stattgefunden haben.
- Überprüfen Sie Ihre Aufrufhäufigkeit (Anfragen pro Sekunde) Funktionen mit bereitgestellter Parallelität haben eine maximale Rate von 10 Anfragen pro Sekunde pro bereitgestellte Parallelität. Beispielsweise kann eine Funktion, die mit 100 bereitgestellter Parallelität konfiguriert ist, 1.000 Anfragen pro Sekunde bearbeiten. Wenn die Aufrufrate 1.000 Anfragen pro Sekunde übersteigt, kann es zu Kaltstarts kommen.

Lambda: Kaltstarts mit neuen Versionen

Problem: Sie erleben Kaltstarts, wenn Sie neue Versionen Ihrer Funktion bereitstellen.

Wenn Sie einen Funktionsalias aktualisieren, verschiebt Lambda die bereitgestellte Parallelität basierend auf den für den Alias konfigurierten Gewichtungen automatisch auf die neue Version.

FehlerKMSDisabledException: Lambda konnte die Umgebungsvariablen nicht entschlüsseln, da der verwendete KMS-Schlüssel deaktiviert ist. Bitte überprüfen Sie die KMS-Schlüsseleinstellungen der Funktion.

Dieser Fehler kann auftreten, wenn Ihr AWS Key Management Service (AWS KMS) -Schlüssel deaktiviert ist oder wenn die Genehmigung, die Lambda die Verwendung des Schlüssels ermöglicht, widerrufen wird. Wenn die Gewährung fehlt, konfigurieren Sie die Funktion so, dass sie einen anderen Schlüssel verwendet. Weisen Sie dann den benutzerdefinierten Schlüssel neu zu, um die Erteilung neu zu erstellen.

Lambda: Unerwarteter Node.js Abbruch in Runtime () Runtime.NodejsExit

Problem: Der Lambda-Runtime-Client hat einen unerwarteten Node.js Exit-Code erkannt.

Dieser Fehler tritt auf, wenn Ihre Funktion beendet wird, bevor alle Versprechen erfüllt sind, z. B. aufgrund eines Codefehlers. Er kann auch auftreten, wenn ein Deadlock Node.js erkannt wird, der verhindert, dass Versprechen abgewickelt werden können. Dieser Fehler betrifft nur Handler im asynchronen Stil, nicht Handler im Callback-Stil.

Betroffene Laufzeiten: 18 und höher. Node.js

Um dieses Problem zu lösen:

1. Überprüfen Sie Ihren Funktionscode auf ungeklärte Versprechen in asynchronen Handlern.
2. Stellen Sie sicher, dass alle Versprechen ordnungsgemäß erfüllt (gelöst oder abgelehnt) wurden, bevor die Funktion abgeschlossen ist.
3. Überprüfen Sie Ihren Code auf mögliche Rennbedingungen bei asynchronen Vorgängen.

Weitere Informationen zu Node.js Exit-Codes und Prozessabbruch finden Sie in der Node.js Dokumentation.

EFS: Die Funktion konnte das EFS-Dateisystem nicht mounten

Fehler:EFSMountFailureException: Die Funktion konnte das EFS-Dateisystem mit dem Access Point arn:aws:elasticfilesystem:us-east- 2:123456789012:access- -015cxmplb72b405fd nicht mounten. point/fsap

Die Mounting-Anforderung der Funktion an das Dateisystem wurde abgelehnt. Überprüfen Sie die Berechtigungen der Funktion und bestätigen Sie, dass das Dateisystem und der Zugriffspunkt vorhanden und einsatzbereit sind.

EFS: Die Funktion konnte keine Verbindung zum EFS-Dateisystem herstellen

FehlerEFSMountConnectivityException: Die Funktion konnte keine Verbindung zum Amazon EFS-Dateisystem mit dem Access Point arn:aws:elasticfilesystem:us-east- 2:123456789012:access --015cxmplb72b405fd herstellen. point/fsap Überprüfen Sie die Netzwerkkonfiguration und versuchen Sie es erneut.

Die Funktion konnte keine Verbindung zum Dateisystem der Funktion mit dem NFS-Protokoll (TCP-Port 2049) herstellen. Überprüfen Sie die Sicherheitsgruppe und die Routingkonfiguration für die Subnetze der VPC.

Wenn Sie diese Fehler nach der Aktualisierung der VPC-Konfigurationseinstellungen Ihrer Funktion erhalten, versuchen Sie, das Dateisystem auszuhängen und erneut auszuhängen.

EFS: Die Funktion konnte das EFS-Dateisystem aufgrund eines Timeouts nicht mounten

FehlerEFSMountTimeoutException: Die Funktion konnte das EFS-Dateisystem mit dem Access Point {arn:aws:elasticfilesystem:us-east- 2:123456789012:access- -015cxmplb72b405fd} aufgrund eines Mount-Timeouts nicht mounten. point/fsap

Die Funktion konnte eine Verbindung mit dem Dateisystem der Funktion herstellen, aber bei der Mounting-Operation trat eine Zeitüberschreitung auf. Versuchen Sie es nach kurzer Zeit erneut und erwägen Sie, die Nebenläufigkeit der Funktion zu begrenzen, um die Belastung des Dateisystems zu reduzieren.

S3-Dateien: Die Funktion konnte das S3-Dateisystem nicht mounten

FehlerS3FilesMountFailureException: Die Funktion konnte das Amazon S3 S3-Dateisystem mit dem Access Point arn:aws:s3files:us-east- 2:123456789012:access- -123456789abcde nicht mounten. point/fsap

Die Mounting-Anforderung der Funktion an das Dateisystem wurde abgelehnt. Überprüfen Sie die Berechtigungen der Funktion und bestätigen Sie, dass das Dateisystem und der Zugriffspunkt vorhanden und einsatzbereit sind.

S3-Dateien: Die Funktion konnte keine Verbindung zum S3-Dateisystem herstellen

FehlerS3FilesMountConnectivityException: Die Funktion konnte keine Verbindung zum Amazon S3 S3-Dateisystem mit dem Access Point arn:aws:s3files:us-east- 2:123456789012:access -

-123456789abcde herstellen. point/fsap Überprüfen Sie die Netzwerkkonfiguration und versuchen Sie es erneut.

Die Funktion konnte keine Verbindung zum Dateisystem der Funktion mit dem NFS-Protokoll (TCP-Port 2049) herstellen. Überprüfen Sie die Sicherheitsgruppe und die Routingkonfiguration für die Subnetze der VPC.

Wenn Sie diese Fehler nach der Aktualisierung der VPC-Konfigurationseinstellungen Ihrer Funktion erhalten, versuchen Sie, das Dateisystem auszuhängen und erneut auszuhängen.

S3-Dateien: Die Funktion konnte das S3-Dateisystem aufgrund eines Timeouts nicht mounten

Fehler:S3FilesMountTimeoutException: Die Funktion konnte das S3-Dateisystem mit dem Access Point {arn:aws:s3files:us-east- 2:123456789012:access- -123456789abcde} aufgrund eines Mount-Timeouts nicht mounten. point/fsap

Die Funktion konnte eine Verbindung mit dem Dateisystem der Funktion herstellen, aber bei der Mounting-Operation trat eine Zeitüberschreitung auf. Versuchen Sie es nach kurzer Zeit erneut und erwägen Sie, die Nebenläufigkeit der Funktion zu begrenzen, um die Belastung des Dateisystems zu reduzieren.

Lambda: Lambda hat einen IO-Prozess erkannt, der zu lange dauerte

EFSIOException: Diese Funktions-Instance wurde angehalten, da Lambda einen I/O-Prozess erkannt hat, der zu lange gedauert hat.

Bei einem vorherigen Aufruf trat eine Zeitüberschreitung auf und Lambda konnte den Funktions-Handler nicht beenden. Dieses Problem kann auftreten, wenn ein angehängtes Dateisystem keine Burst-Credits hat und der Basisdurchsatz nicht ausreicht. Um den Durchsatz zu erhöhen, können Sie die Größe des Dateisystems erhöhen oder den bereitgestellten Durchsatz verwenden.

Container CodeArtifactUserException : Fehler

Fehler: CodeArtifactUserPendingException Fehlermeldung

Die Optimierung CodeArtifact steht noch aus. Die Funktion wechselt in den aktiven Zustand, wenn Lambda die Optimierung abgeschlossen hat. HTTP-Antwortcode 409.

Fehler: CodeArtifactUserDeletedException Fehlermeldung

Das CodeArtifact soll voraussichtlich gelöscht werden. HTTP-Antwortcode 409.

Fehler: CodeArtifactUserFailedException Fehlermeldung

Lambda konnte den Code nicht optimieren. Sie müssen den Code korrigieren und erneut hochladen. HTTP-Antwortcode 409.

Container: InvalidEntrypoint Fehler

Fehler: Runtime.ExitError oder „errorType“: "" Runtime.InvalidEntrypoint

Stellen Sie sicher, dass der ENTRYPOINT zu Ihrem Container-Image den absoluten Pfad als Standort enthält. Stellen Sie außerdem sicher, dass das Image keinen Symlink als ENTRYPOINT enthält.

Fehler: Sie verwenden eine CloudFormation Vorlage und Ihr Container-ENTRYPOINT wird mit einem Nullwert oder einem leeren Wert überschrieben.

Überprüfen Sie die ImageConfigRessource in der Vorlage. CloudFormation Wenn Sie eine `ImageConfig`-Ressource in Ihrer Vorlage deklarieren, müssen Sie nichtleere Werte für alle drei Eigenschaften angeben.

Beheben von Ausführungsproblemen in Lambda

Wenn die Lambda-Laufzeit Ihren Funktionscode ausführt, wird das Ereignis möglicherweise auf einer Instanz der Lambda-Ereignisfunktion verarbeitet, die seit einiger Zeit Ereignisse verarbeitet, oder es muss möglicherweise eine neue Instanz initialisiert werden. Fehler können während der Funktionsinitialisierung, wenn Ihr Handler-Code das Ereignis verarbeitet oder wenn Ihre Funktion eine Antwort zurückgibt (oder nicht zurückgibt), auftreten.

Funktionsausführungsfehler können durch Probleme mit Code, Funktionskonfiguration, nachgeschalteten Ressourcen oder Berechtigungen verursacht werden. Wenn Sie Ihre Funktion direkt aufrufen, sehen Sie Funktionsfehler in der Antwort von Lambda. Wenn Sie Ihre Funktion asynchron mit einer Ereignisquellen-Zuweisung oder über einen anderen Service aufrufen, finden Sie möglicherweise Fehler in Protokollen, eine Warteschlange für unzustellbare Nachrichten oder ein Ziel bei Ausfall. Die Optionen für die Fehlerbehandlung und das Wiederholungsverhalten variieren je nachdem, wie Sie Ihre Funktion aufrufen, und nach der Art des Fehlers.

Wenn Ihr Funktionscode oder die Lambda-Laufzeit einen Fehler zurückgibt, ist der Statuscode in der Antwort von Lambda „200 OK“. Das Vorhandensein eines Fehlers in der Antwort wird durch einen

Header namens `X-Amz-Function-Error` angezeigt. Statuscodes der Serien 400 und 500 sind für Aufruffehler reserviert.

Themen

- Lambda: Remote-Debugging mit Visual Studio Code
- Lambda: Die Ausführung dauert zu lange
- Lamda: Unerwartete Ereignis-Nutzdaten
- Lambda: Unerwartet große Nutzdaten
- Lambda: Fehler beim Kodieren und Dekodieren von JSON
- Lambda: Protokolle oder Ablaufverfolgungen erscheinen nicht
- Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt
- Lambda: Die Funktion kehrt zurück, bevor die Ausführung beendet ist
- Lamda: Ausführen einer unbeabsichtigten Funktionsversion oder eines Alias
- Lambda: Erkennen von Endlosschleifen
- Allgemein: Nichtverfügbarkeit nachgelagerter Dienste
- AWS SDK: Versionen und Updates
- Python: Bibliotheken werden falsch geladen
- Java: Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde
- Kafka: Probleme bei der Fehlerbehandlung und beim erneuten Versuch der Konfiguration

Lambda: Remote-Debugging mit Visual Studio Code

Problem: Schwierigkeiten bei der Behebung des komplexen Verhaltens von Lambda-Funktionen in der tatsächlichen Umgebung AWS

Lambda bietet ein Remote-Debugging-Feature über das AWS Toolkit for Visual Studio Code. Anweisungen zur Einrichtung und allgemeinen Nutzung finden Sie unter Ferndebuggen von Lambda-Ereignisfunktionen mit Visual Studio Code.

Ausführliche Anweisungen zur Fehlerbehebung, zu erweiterten Anwendungsfällen und zur regionalen Verfügbarkeit finden Sie unter Lambda-Funktionen per Fernzugriff im AWS Toolkit for Visual Studio Code Benutzerhandbuch.

Lambda: Die Ausführung dauert zu lange

Problem: Die Ausführung der Funktion dauert zu lange.

Wenn die Ausführung Ihres Codes in Lambda viel länger dauert als auf Ihrem lokalen Computer, ist er möglicherweise durch den für die Funktion verfügbaren Speicher oder die Rechenleistung eingeschränkt. Konfigurieren Sie die Funktion mit zusätzlichem Arbeitsspeicher, um sowohl Arbeitsspeicher als auch CPU zu erhöhen.

Lamda: Unerwartete Ereignis-Nutzdaten

Problem: Funktionsfehler im Zusammenhang mit fehlerhaftem JSON oder unzureichender Datenvalidierung

Alle Lambda-Funktionen erhalten eine Ereignis-Nutzdaten im ersten Parameter des Handlers. Die Event-Payload ist eine JSON-Struktur, die Arrays und verschachtelte Elemente enthalten kann.

Missgebildetes JSON kann auftreten, wenn es von vorgelagerten Diensten bereitgestellt wird, die kein robustes Verfahren zur Überprüfung von JSON-Strukturen verwenden. Dies tritt auf, wenn Dienste Textzeichenfolgen miteinander verknüpfen oder Benutzereingaben einbetten, die nicht bereinigt wurden. JSON wird auch häufig für die Weitergabe zwischen Diensten serialisiert. Analysieren Sie JSON-Strukturen immer sowohl als Produzent als auch als Nutzer von JSON, um sicherzustellen, dass die Struktur gültig ist.

Ebenso kann es zu Fehlern kommen, wenn nicht auf Wertebereiche in den Ereignis-Nutzdaten geprüft wird. Eine Funktion, die an den Wert des Handlers übergeben wird:

```
exports.handler = async (event) => {
    let pct = event.taxPct
    let salary = event.salary

    // Calculate % of paycheck for taxes
    return (salary * pct)
}
```

Diese Funktion verwendet ein Gehalt und einen Steuersatz aus den Ereignis-Nutzdaten, um die Berechnung durchzuführen. Der Code überprüft jedoch nicht, ob die Attribute vorhanden sind. Außerdem werden Datentypen nicht überprüft oder Grenzen nicht gewährleistet, z. B. die Sicherstellung, dass der Steuersatz zwischen 0 und 1 liegt. Daher führen Werte außerhalb dieser Grenzen zu unsinnigen Ergebnissen. Ein falscher Typ oder ein fehlendes Attribut führt zu einem Laufzeitfehler.

Erstellen Sie Tests, um sicherzustellen, dass Ihre Funktion größere Nutzdaten verarbeitet. Die maximale Größe für eine Lambda-Ereignis-Nutzlast beträgt 1 MB. Je nach Inhalt können größere Payloads bedeuten, dass mehr Elemente an die Funktion übergeben werden oder mehr Binärdaten in ein JSON-Attribut eingebettet sind. In beiden Fällen kann dies zu mehr Verarbeitung für eine Lambda-Funktion führen.

Größere Nutzdaten können auch zu Timeouts führen. Eine Lambda-Funktion verarbeitet beispielsweise einen Datensatz pro 100 ms und hat ein Timeout von 3 Sekunden. Die Verarbeitung ist für 0-29 Elemente in den Nutzdaten erfolgreich. Sobald die Nutzdaten jedoch mehr als 30 Elemente enthalten, bricht die Funktion ab und gibt einen Fehler aus. Um dies zu vermeiden, stellen Sie sicher, dass die Timeouts so eingestellt sind, dass sie die zusätzliche Verarbeitungszeit für die maximale Anzahl erwarteter Elemente berücksichtigen.

Lambda: Unerwartet große Nutzdaten

Problem: Funktionen verursachen Timeouts und Fehler aufgrund großer Nutzdaten

Größere Nutzdaten können zu Timeouts und Fehlern führen. Wir empfehlen, Tests zu erstellen, um sicherzustellen, dass Ihre Funktion die größten zu erwartenden Nutzdaten verarbeiten kann, und zu prüfen, ob das Zeitlimit für die Funktion richtig eingestellt ist.

Darüber hinaus können bestimmte Ereignis-Nutzdaten Verweise auf andere Ressourcen enthalten. Beispielsweise kann eine Lambda-Funktion mit 128 MB Speicher eine Bildverarbeitung für eine JPG-Datei durchführen, die als Objekt in S3 gespeichert ist. Die Funktion funktioniert erwartungsgemäß mit kleineren Bilddateien.

Wenn jedoch eine größere JPG-Datei als Eingabe bereitgestellt wird, gibt die Lambda-Funktion einen Fehler aus, da nicht genügend Speicherplatz zur Verfügung steht. Um dies zu vermeiden, sollten die Testfälle Beispiele aus dem oberen Bereich der erwarteten Datengrößen enthalten. Der Code sollte auch die Nutzdatengröße validieren.

Lambda: Fehler beim Kodieren und Dekodieren von JSON

Problem: NoSuchKey Ausnahme beim Parsen von JSON-Eingaben.

Stellen Sie sicher, dass Sie JSON-Attribute korrekt verarbeiten. Für Ereignisse, die von S3 generiert wurden, enthält das Attribut `s3.object.key` beispielsweise einen URL-codierten Objektschlüsselnamen. Viele Funktionen verarbeiten dieses Attribut als Text, um das referenzierte S3-Objekt zu laden:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: event.Records[0].s3.object.key
}).promise()
```

Dieser Code funktioniert mit dem Schlüsselnamen `james.jpg`, löst aber einen `NoSuchKey`-Fehler aus, wenn der Name `james beswick.jpg` lautet. Da die URL-Kodierung Leerzeichen und andere Zeichen in einem Schlüsselnamen konvertiert, müssen Sie sicherstellen, dass Funktionen Schlüssel dekodieren, bevor Sie diese Daten verwenden:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "))
}).promise()
```

Lambda: Protokolle oder Ablaufverfolgungen erscheinen nicht

Problem: Protokolle werden nicht in CloudWatch Logs angezeigt.

Problem: Spuren werden nicht in angezeigt AWS X-Ray.

Ihre Funktion benötigt die Erlaubnis, CloudWatch Logs und aufzurufen X-Ray. Aktualisieren Sie die Ausführungsrolle, um ihr die Berechtigung zu erteilen. Fügen Sie die folgenden verwalteten Richtlinien hinzu, um Protokolle und Ablaufverfolgung zu aktivieren.

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

Wenn Sie Ihrer Funktion Berechtigungen hinzufügen, führen Sie auch eine triviale Aktualisierung ihres Codes oder ihrer Konfiguration durch. Dies zwingt ausgeführte Instances Ihrer Funktion, die veraltete Anmeldeinformationen haben, anzuhalten und ersetzt zu werden.

> **Note**
>
> Es kann 5 bis 10 Minuten dauern, bis Logs nach einem Funktionsaufruf angezeigt werden.

Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt

Problem: Funktionsprotokolle fehlen in den CloudWatch Protokollen, obwohl meine Berechtigungen korrekt sind

Wenn Ihr System die Kontingentgrenzen für CloudWatch Logs AWS-Konto erreicht, wird die Funktionsprotokollierung CloudWatch gedrosselt. In diesem Fall werden einige der von Ihren Funktionen ausgegebenen Protokolle möglicherweise nicht in CloudWatch den Protokollen angezeigt.

Wenn Ihre Funktion Logs mit einer zu hohen Rate ausgibt, als dass Lambda sie verarbeiten könnte, kann dies auch dazu führen, dass Protokollausgaben nicht in CloudWatch Logs erscheinen. Wenn Lambda Protokolle nicht mit der Geschwindigkeit senden kann, CloudWatch an die Ihre Funktion sie generiert, werden Protokolle gelöscht, um zu verhindern, dass die Ausführung Ihrer Funktion verlangsamt wird. Gehen Sie davon aus, dass gelöschte Protokolle konsistent beobachtet werden, wenn Ihr Protokolldurchsatz MB/s für einen einzelnen Protokollstream 2 überschreitet.

Wenn Ihre Funktion für die Verwendung von Protokollen im JSON-Format konfiguriert ist, versucht Lambda, beim Löschen von Protokollen ein `logsDropped`Ereignis an CloudWatch Logs zu senden. Wenn jedoch die Protokollierung Ihrer Funktion CloudWatch gedrosselt wird, erreicht dieses Ereignis möglicherweise nicht CloudWatch Logs, sodass Sie nicht immer einen Datensatz sehen, wenn Lambda Logs löscht.

Gehen Sie wie folgt vor, um zu überprüfen, ob Ihr System die Kontingentgrenzen für CloudWatch Logs erreicht AWS-Konto hat:

1. Öffnen Sie die Service Quotas-Konsole.
2. Wählen Sie im Navigationsbereich AWS -Services.
3. Suchen Sie in der AWS Serviceliste nach Amazon CloudWatch Logs.
4. Wählen Sie in der Liste mit den Service Quotas die Kontingente `CreateLogGroup throttle limit in transactions per second`, `CreateLogStream throttle limit in transactions per second` und `PutLogEvents throttle limit in transactions per second` aus, um die Auslastung anzuzeigen.

Sie können auch CloudWatch Alarme einrichten, die Sie benachrichtigen, wenn Ihre Kontoauslastung ein von Ihnen für diese Kontingente festgelegtes Limit überschreitet. Weitere Informationen finden Sie unter Einen CloudWatch Alarm auf der Grundlage eines statischen Schwellenwerts erstellen.

Wenn die standardmäßigen Kontingentgrenzen für CloudWatch Logs für Ihren Anwendungsfall nicht ausreichen, können Sie eine Erhöhung des Kontingents beantragen.

Lambda: Die Funktion kehrt zurück, bevor die Ausführung beendet ist

Problem: (Node.js) Die Funktion kehrt zurück, bevor die Ausführung des Codes abgeschlossen ist

Viele Bibliotheken, einschließlich des AWS SDK, arbeiten asynchron. Wenn Sie einen Netzwerkaufruf tätigen oder einen anderen Vorgang ausführen, für den auf eine Antwort gewartet werden muss, geben Bibliotheken ein Objekt zurück, das als Zusage bezeichnet wird und mit dem der Status der Operation im Hintergrund nachverfolgt wird.

Um zu warten, bis die Zusage in eine Antwort aufgelöst wird, verwenden Sie das Schlüsselwort `await`. Dadurch wird verhindert, dass der Handler-Code ausgeführt wird, bis die Zusage in ein Objekt aufgelöst wird, das die Antwort enthält. Wenn Sie die Daten aus der Antwort in Ihrem Code nicht verwenden müssen, können Sie die Zusage direkt an die Laufzeit zurückgeben.

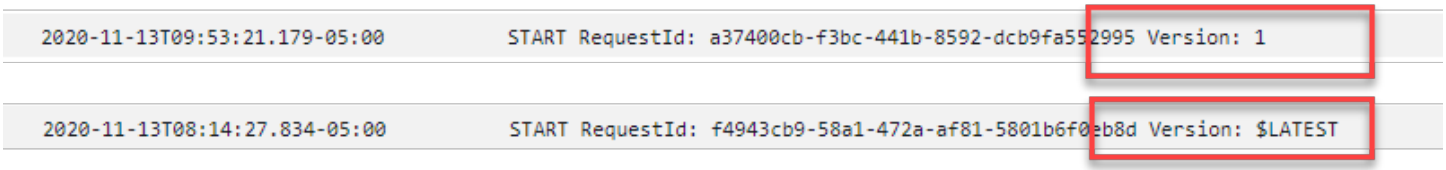
Einige Bibliotheken geben keine Zusagen zurück, können aber in Code verpackt werden, der dies tut. Weitere Informationen finden Sie unter Definieren Sie den Lambda-Ereignisfunktionshandler in Node.js.

Lamda: Ausführen einer unbeabsichtigten Funktionsversion oder eines Alias

Problem: Funktionsversion oder Alias wurde nicht aufgerufen

Wenn Sie neue Lambda-Funktionen in der Konsole veröffentlichen oder verwenden AWS SAM, wird die neueste Codeversion durch `$LATEST` dargestellt. Standardmäßig zielen Aufrufe, die keine Version oder keinen Alias angeben, automatisch auf die `$LATEST`-Version Ihres Funktionscodes ab.

Wenn Sie bestimmte Funktionsversionen oder Aliase verwenden, handelt es sich dabei um unveränderliche veröffentlichte Versionen einer Funktion zusätzlich zu `$LATEST`. Stellen Sie bei der Problembehebung dieser Funktionen zunächst fest, ob der Aufrufer die beabsichtigte Version oder den Alias aufgerufen hat. Überprüfen Sie dazu Ihre Funktionsprotokolle. Die Version der aufgerufenen Funktion wird immer in der START-Protokollzeile angezeigt:

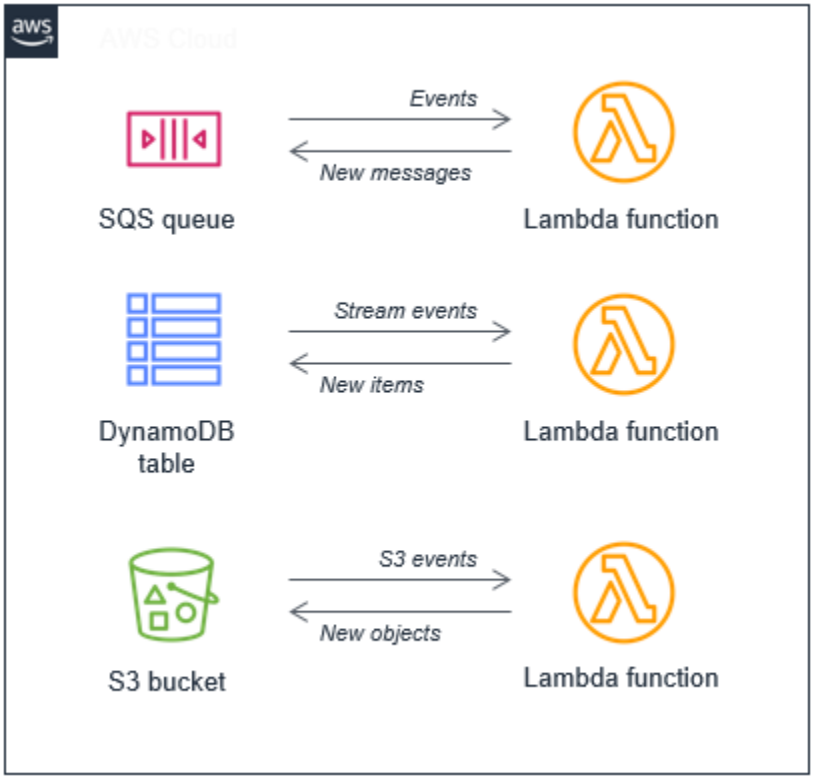


Lambda: Erkennen von Endlosschleifen

Problem: Endlosschleifenmuster im Zusammenhang mit Lambda-Funktionen

Es gibt zwei Arten von Endlosschleifen in Lambda-Funktionen. Die erste befindet sich innerhalb der Funktion selbst und wird durch eine Schleife verursacht, die niemals beendet wird. Der Aufruf endet erst, wenn die Funktion ein Timeout verursacht. Sie können diese identifizieren, indem Sie Timeouts überwachen und dann das Schleifenverhalten beheben.

Die zweite Art von Schleife besteht zwischen Lambda-Funktionen und anderen AWS Ressourcen. Diese treten auf, wenn ein Ereignis aus einer Ressource wie einem S3-Bucket eine Lambda-Funktion aufruft, die dann mit derselben Quellressource interagiert, um ein weiteres Ereignis auszulösen. Dadurch wird die Funktion erneut aufgerufen, wodurch eine weitere Interaktion mit demselben S3-Bucket erstellt wird, und so weiter. Diese Arten von Schleifen können durch eine Reihe verschiedener AWS Ereignisquellen verursacht werden, darunter Amazon SQS SQS-Warteschlangen und DynamoDB-Tabellen. Sie können diese Muster mithilfe der rekursiven Schleifenerkennung identifizieren.



Sie können diese Schleifen vermeiden, indem Sie sicherstellen, dass Lambda-Funktionen in Ressourcen schreiben, die nicht mit der verbrauchenden Ressource identisch sind. Wenn Sie Daten

wieder auf der verbrauchenden Ressource veröffentlichen müssen, stellen Sie sicher, dass die neuen Daten nicht dasselbe Ereignis auslösen. Verwenden Sie alternativ die Ereignisfilterung. Hier sind beispielsweise zwei Lösungsvorschläge für Endlosschleifen mit S3- und DynamoDB-Ressourcen:

- Wenn Sie in denselben S3-Bucket zurückschreiben, verwenden Sie ein anderes Präfix oder Suffix als beim Ereignisauslöser.
- Wenn Sie Elemente in dieselbe DynamoDB-Tabelle schreiben, fügen Sie ein Attribut hinzu, nach dem eine verbrauchende Lambda-Funktion filtern kann. Wenn Lambda das Attribut findet, führt dies nicht zu einem weiteren Aufruf.

Allgemein: Nichtverfügbarkeit nachgelagerter Dienste

Problem: Die nachgelagerten Dienste, auf die Ihre Lambda-Funktion angewiesen ist, sind nicht verfügbar

Stellen Sie bei Lambda-Funktionen, die Drittanbieter-Endpunkte oder andere nachgelagerte Ressourcen aufrufen, sicher, dass sie Servicefehler und Timeouts behandeln können. Diese nachgelagerten Ressourcen können unterschiedliche Reaktionszeiten haben oder aufgrund von Serviceunterbrechungen nicht verfügbar sein. Je nach Implementierung können diese Downstream-Fehler als Lambda-Timeouts oder Ausnahmen auftreten, wenn die Fehlerantwort des Dienstes nicht im Funktionscode behandelt wird.

Implementieren Sie eine geeignete Fehlerbehandlung und Wiederholungslogik, wenn eine Funktion von einem nachgelagerten Dienst abhängt, z. B. von einem API-Aufruf. Für kritische Dienste sollte die Lambda-Funktion Metriken oder Protokolle veröffentlichen. CloudWatch Wenn beispielsweise eine Zahlungs-API eines Drittanbieters nicht mehr verfügbar ist, kann die Lambda-Funktion diese Informationen protokollieren. Sie können dann CloudWatch Alarme einrichten, um Benachrichtigungen zu diesen Fehlern zu senden.

Da Lambda schnell skaliert werden kann, könnten Downstream-Dienste, die nicht serverlos sind, Schwierigkeiten haben, Datenverkehrsspitzen zu bewältigen. Es gibt drei gängige Ansätze, um damit umzugehen:

- Caching: Erwägen Sie, die Ergebnisse von Werten, die von Drittanbieterdiensten zurückgegeben werden, zwischenzuspeichern, wenn diese sich nicht häufig ändern. Sie können diese Werte in einer globalen Variablen in Ihrer Funktion oder einem anderen Dienst speichern. Beispielsweise könnten die Ergebnisse einer Produktlistenabfrage von einer Amazon RDS-Instance innerhalb

der Funktion für einen bestimmten Zeitraum gespeichert werden, um redundante Abfragen zu vermeiden.

- Warteschlangen: Fügen Sie beim Speichern oder Aktualisieren von Daten eine Amazon-SQS-Warteschlange zwischen der Lambda-Funktion und der Ressource hinzu. In der Warteschlange werden Daten dauerhaft gespeichert, während der nachgelagerte Dienst Nachrichten verarbeitet.
- Proxys: Wenn in der Regel langlebige Verbindungen verwendet werden, z. B. für Amazon-RDS-Instances, verwenden Sie eine Proxyebene, um diese Verbindungen zu bündeln und wiederzuverwenden. Für relationale Datenbanken ist Amazon RDS Proxy ein Service, der dazu beitragen soll, die Skalierbarkeit und Stabilität von Anwendungen zu verbessern. Lambda-based

AWS SDK: Versionen und Updates

Problem: Das in der Runtime enthaltene AWS SDK ist nicht die neueste Version

Problem: Das in der Runtime enthaltene AWS SDK wird automatisch aktualisiert

Zu den Laufzeiten für interpretierte Sprachen gehört eine Version des AWS SDK. Lambda aktualisiert diese Laufzeiten regelmäßig, um die neueste SDK-Version zu verwenden. Die in Ihrer Laufzeit enthaltene SDK-Version finden Sie in den folgenden Abschnitten:

- Runtime umfasste SDK-Versionen () Node.js
- In der Laufzeit enthaltene SDK-Versionen (Python)
- In der Laufzeit enthaltene SDK-Versionen (Ruby)

Um eine neuere Version des AWS SDK zu verwenden oder Ihre Funktionen an eine bestimmte Version zu binden, können Sie die Bibliothek mit Ihrem Funktionscode bündeln oder eine Lambda-Schicht erstellen. Weitere Informationen zum Erstellen eines Bereitstellungspakets mit Abhängigkeiten finden Sie in den folgenden Themen:

Node.js

Stellen Sie Node.js Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit

Python

Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions

Ruby

Stellen Sie Ruby Lambda Event Functions mit.zip-Dateiarchiven bereit

Java

Stellen Sie Java Lambda Event Functions mit.zip- oder JAR-Dateiarchiven bereit

Go

Stellen Sie Go Lambda Event Functions mit.zip-Dateiarchiven bereit

C#

C#-Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven erstellen und bereitstellen

PowerShell

Stellen Sie PowerShell Lambda-Ereignisfunktionen mit ZIP-Dateiarchiven bereit

Python: Bibliotheken werden falsch geladen

Problem: (Python) Einige Bibliotheken werden nicht korrekt aus dem Bereitstellungspaket geladen

Bibliotheken mit Erweiterungsmodulen, die in C oder C ++ geschrieben sind, müssen in einer Umgebung mit derselben Prozessorarchitektur wie Lambda (Amazon Linux) kompiliert werden. Weitere Informationen finden Sie unter Arbeiten mit ZIP-Dateiarchiven für Python Lambda Event Functions.

Java: Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde

Problem: (Java) Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde

Optimieren Sie Ihren Compiler mithilfe des `JAVA_TOOL_OPTIONS`-Parameters. Lambda-Laufzeiten für Java 17 und spätere Java-Versionen ändern die Standard-Compileroptionen. Die Änderung verbessert die Kaltstartzeiten für kurzlebige Funktionen, aber das bisherige Verhalten ist besser für rechenintensive, länger laufende Funktionen geeignet. Setzen Sie `JAVA_TOOL_OPTIONS` auf `-XX:-TieredCompilation`, um zum Verhalten von Java 11 zurückzukehren. Weitere Informationen zum Parameter `JAVA_TOOL_OPTIONS` erhalten Sie unter the section called “Grundlegendes zur Umgebungsvariablen `JAVA_TOOL_OPTIONS`”.

Kafka: Probleme bei der Fehlerbehandlung und beim erneuten Versuch der Konfiguration

Problem: Bei der Zuordnung von Kafka-Ereignisquellen können Einstellungen für Wiederholungsversuche oder Ziele für den Fall eines Fehlers nicht konfiguriert werden

Kafka-Wiederholungskonfigurationen und Ziele für den Fall eines Fehlers sind nur für Zuordnungen von Ereignisquellen verfügbar, bei denen der Bereitstellungsmodus aktiviert ist. Stellen Sie sicher, dass Sie Ihre Konfiguration vorgenommen haben, `ProvisionedPollerConfig` bevor Sie versuchen, `MinimumPollers` Wiederholungskonfigurationen einzurichten.

Häufige Konfigurationsfehler:

- Unendliche Wiederholungen mit Bisect Batch — Sie können die Option nicht aktivieren`BisectBatchOnFunctionError`, wenn die Option auf -1 (unendlich) gesetzt `MaximumRetryAttempts` ist. Legen Sie ein begrenztes Limit für Wiederholungsversuche fest oder deaktivieren Sie Bisect Batch.
- Rekursion zum selben Thema — Das Kafka-Zielthema bei einem Fehler darf nicht mit einem Ihrer Quellthemen identisch sein. Wählen Sie einen anderen Themennamen für Ihr aktuelles Thema.
- Ungültiges Kafka-Zielformat — Verwenden Sie das `kafka://<topic-name>` Format, wenn Sie ein Kafka-Thema als Ziel für den Fall eines Fehlers angeben.
- kafka: WriteData Berechtigungsprobleme — Stellen Sie sicher, dass Ihre Ausführungsrolle über `kafka-cluster:WriteData` Berechtigungen für das Zielthema verfügt. Das Thema existiert nicht, Timeout-Ausnahmen oder Probleme mit der Einschränkung der Schreib-API erfordern möglicherweise eine Erhöhung der Kontolimits.

Behebung von Problemen mit der Zuordnung von Ereignisquellen in Lambda

Probleme in Lambda, die sich auf die Zuordnung von Ereignisquellen beziehen, können komplexer sein, da sie das Debuggen mehrerer Services beinhalten. Darüber hinaus kann das Verhalten der Ereignisquelle je nach der verwendeten Ereignisquelle unterschiedlich sein. In diesem Abschnitt werden häufig auftretende Probleme im Zusammenhang mit der Zuordnung von Ereignisquellen aufgeführt und Hinweise zu deren Identifizierung und Behebung gegeben.

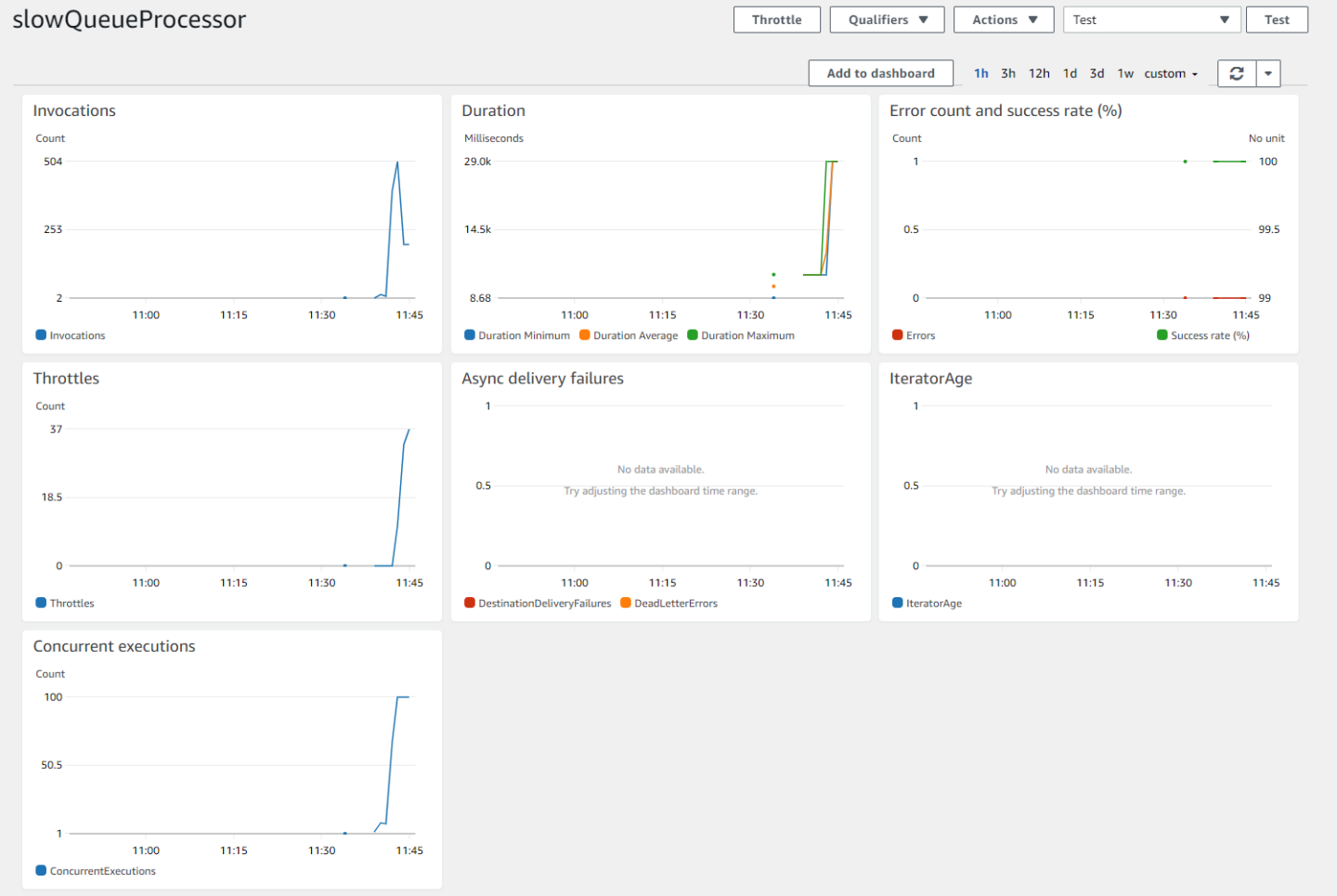
> **Note**
>
> In diesem Abschnitt wird zur Veranschaulichung eine Amazon SQS SQS-Ereignisquelle verwendet. Die Prinzipien gelten jedoch auch für andere Ereignisquellenzuordnungen, die Nachrichten für Lambda-Ereignisfunktionen in die Warteschlange stellen.

Drosselung erkennen und verwalten

In Lambda erfolgt die Drosselung, wenn das Gleichzeitigkeitslimit Ihrer Funktion oder Ihres Kontos erreicht ist. Betrachten Sie das folgende Beispiel, in dem es eine Lambda-Funktion gibt, um Nachrichten aus einer Amazon-SQS-Warteschlange zu lesen. Diese Lambda-Funktion simuliert 30-Sekunden-Aufrufe und hat eine Batchgröße von 1. Das bedeutet, dass die Funktion alle 30 Sekunden nur eine Nachricht verarbeitet:

```
const doWork = (ms) => new Promise(resolve => setTimeout(resolve, ms))

exports.handler = async (event) => {
    await doWork(30000)

}
```

Bei einer so langen Aufrufzeit kommen Nachrichten schneller in der Warteschlange an als sie verarbeitet werden. Wenn die nicht reservierte Gleichzeitigkeit Ihres Kontos 100 beträgt, skaliert Lambda auf 100 gleichzeitige Ausführungen hoch, und dann erfolgt eine Drosselung. Sie können dieses Muster in den CloudWatch Metriken für die Funktion erkennen:



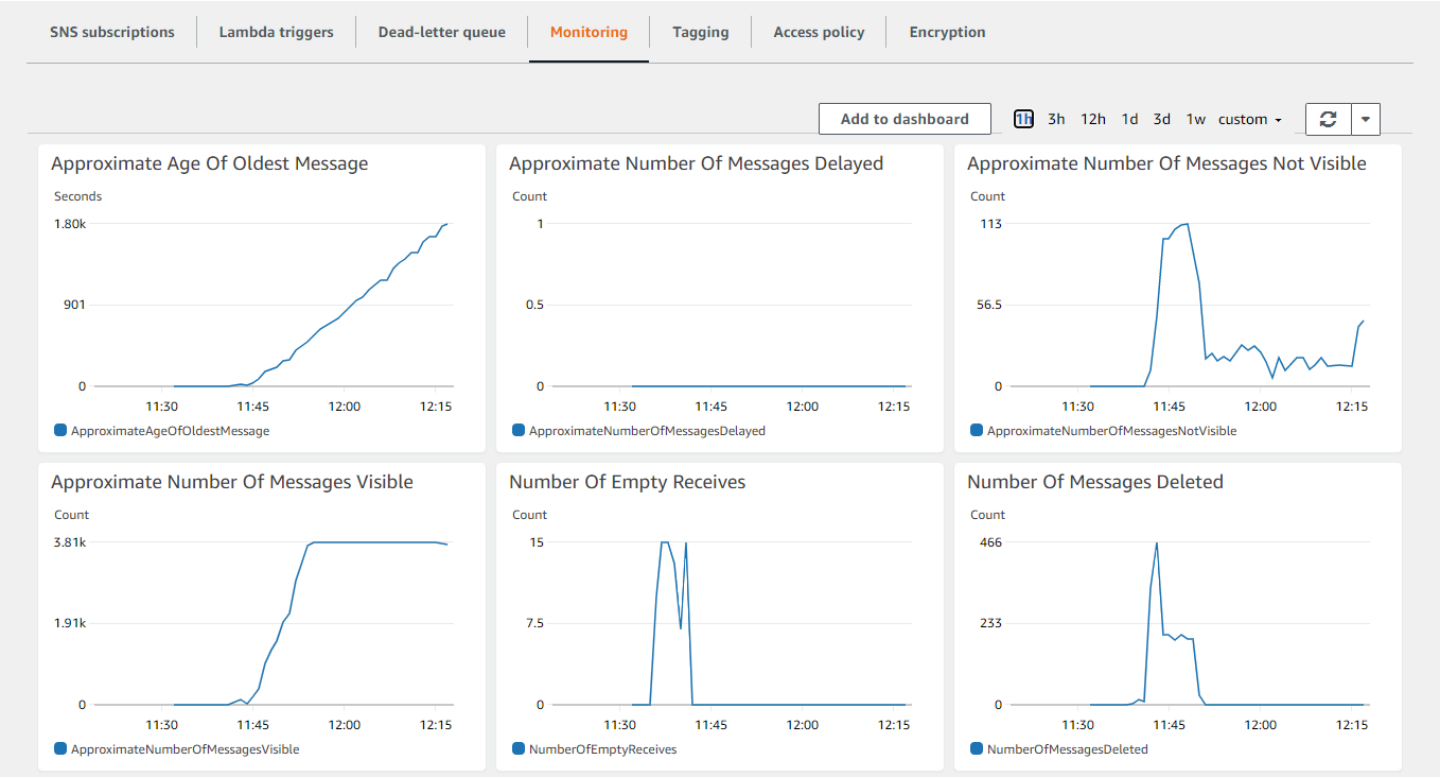
CloudWatch Die Metriken für die Funktion zeigen keine Fehler, aber das Diagramm Gleichzeitige Ausführungen zeigt, dass die maximale Parallelität von 100 erreicht ist. Aus diesem Grund zeigt das Diagramm Drosselungen, dass die Drosselung bereits besteht.

Sie können Drosselungen anhand von CloudWatch Alarmen erkennen und immer dann einen Alarm einstellen, wenn die Drosselungsmetrik für eine Funktion größer als 0 ist. Nachdem Sie das Drosselungsproblem identifiziert haben, stehen Ihnen einige Lösungsmöglichkeiten zur Verfügung:

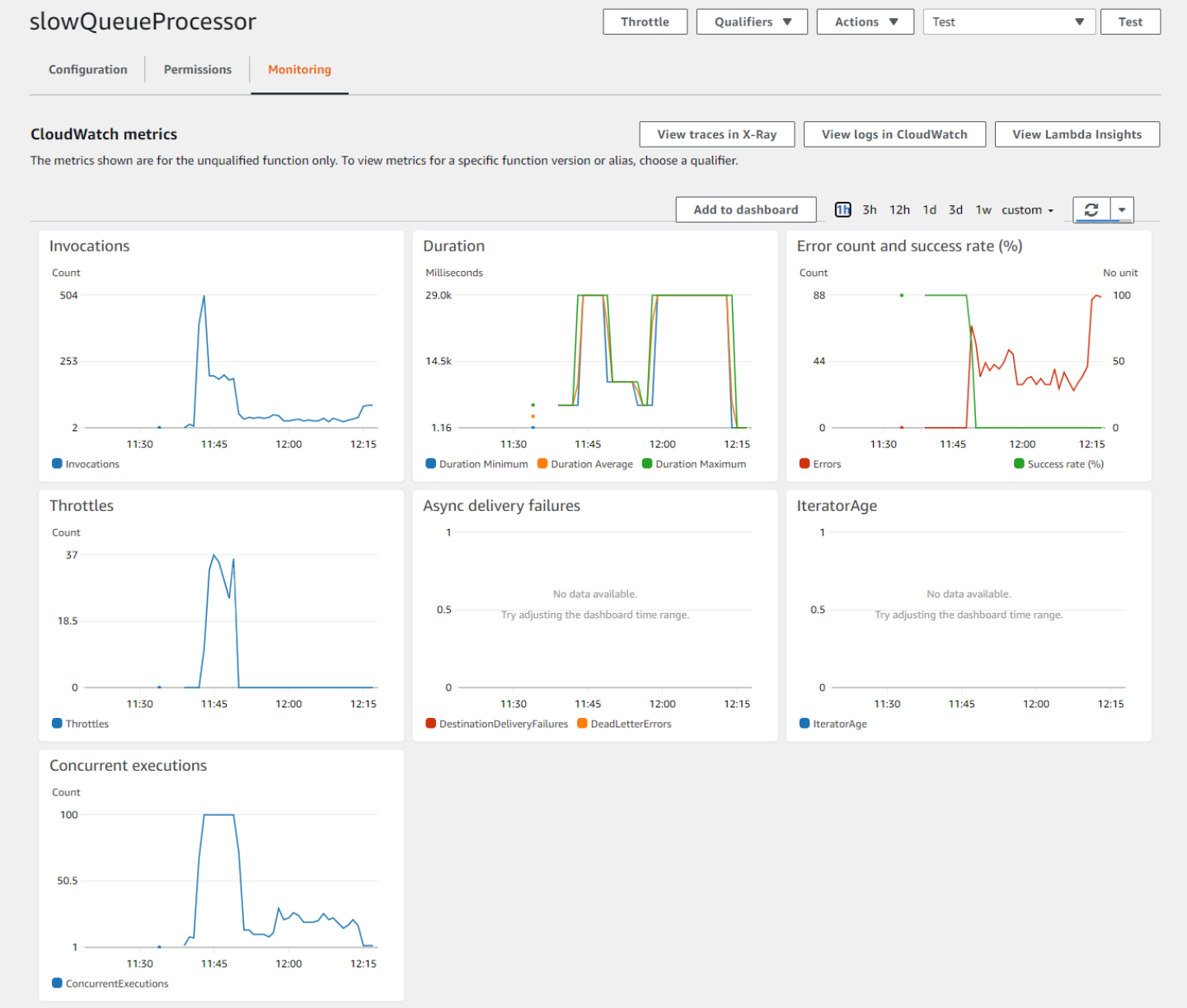
- Fordern Sie beim AWS Support in dieser Region eine Erhöhung der Parallelität an.
- Identifizieren Sie Leistungsprobleme in der Funktion, um die Verarbeitungsgeschwindigkeit und damit den Durchsatz zu verbessern.
- Erhöhen Sie die Batchgröße der Funktion, sodass bei jedem Aufruf mehr Nachrichten verarbeitet werden.

Fehler in der Verarbeitungsfunktion

Wenn die Verarbeitungsfunktion Fehler ausgibt, gibt Lambda die Nachrichten an die SQS-Warteschlange zurück. Lambda verhindert, dass Ihre Funktion skaliert wird, um Skalierungsfehler zu vermeiden. Die folgenden SQS-Metriken in CloudWatch deuten auf ein Problem mit der Warteschlangenverarbeitung hin:

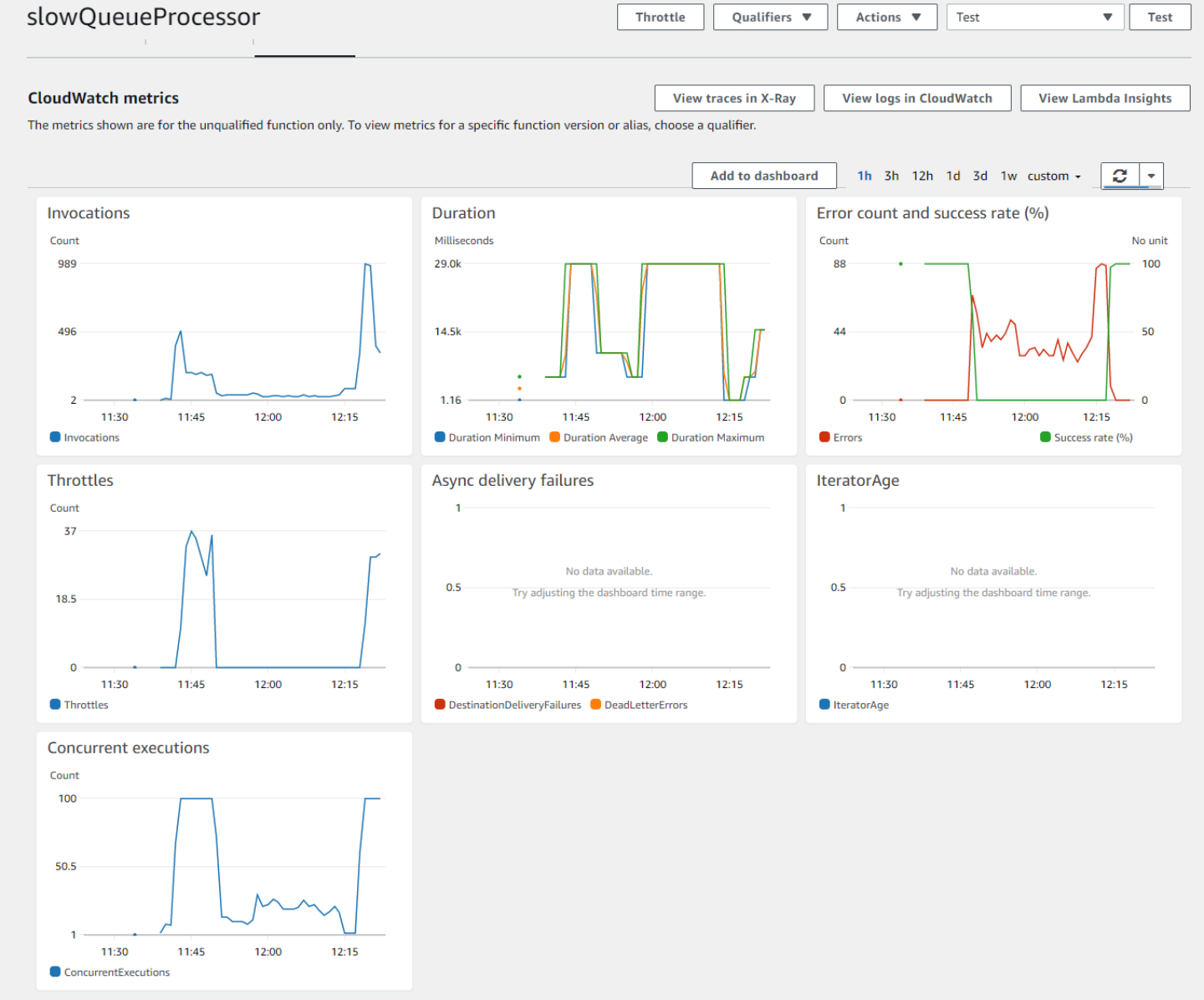


Insbesondere nehmen sowohl das Alter der ältesten Nachricht als auch die Anzahl der sichtbaren Nachrichten zu, während keine Nachrichten gelöscht werden. Die Warteschlange wächst weiter, aber Nachrichten werden nicht verarbeitet. Die CloudWatch Metriken für die Verarbeitung der Lambda-Funktion weisen auch darauf hin, dass ein Problem vorliegt:



Die Metrik für die Anzahl der Fehler ist ungleich Null und nimmt weiter zu, während die Anzahl der gleichzeitigen Ausführungen zurückgegangen ist und die Drosselung eingestellt wurde. Dies deutet darauf hin, dass Lambda die Hochskalierung Ihrer Funktion aufgrund von Fehlern eingestellt hat. Die CloudWatch Protokolle für die Funktion enthalten Einzelheiten zur Art des Fehlers.

Sie können dieses Problem lösen, indem Sie die Funktion identifizieren, die den Fehler verursacht hat und dann den Fehler finden und beheben. Nachdem Sie den Fehler behoben und den neuen Funktionscode bereitgestellt haben, sollten die CloudWatch Messwerte zeigen, dass die Verarbeitung wiederhergestellt ist:



Hier sinkt die Metrik Anzahl der Fehler auf Null und die Metrik Erfolgsquote kehrt auf 100 % zurück. Lambda beginnt erneut mit der Hochskalierung der Funktion, wie im Diagramm Gleichzeitige Ausführungen dargestellt.

Identifizierung und Behandlung von Gegendruck

Wenn ein Ereignisproduzent konsistent Nachrichten für eine SQS-Warteschlange schneller generiert, als eine Lambda-Funktion diese verarbeiten kann, entsteht Gegendruck. In diesem Fall sollte die SQS-Überwachung das Alter der ältesten Nachricht linear ansteigen lassen, zusammen mit der ungefähren Anzahl der sichtbaren Nachrichten. Mithilfe von Alarmen können Sie Gegendruck in Warteschlangen erkennen. CloudWatch

Die Schritte zur Behebung von Gegendruck hängen von Ihrem Workload ab. Wenn das Hauptziel darin besteht, die Verarbeitungskapazität und den Durchsatz durch die Lambda-Funktion zu erhöhen, haben Sie einige Möglichkeiten:

- Beantragen Sie über die Servicekontingenten-Konsole eine Erhöhung, wenn Ihre Funktion die Parallelitätsquote Ihres Kontos erreicht.
- Aktivieren Sie den Bereitstellungsmodus, um die Zuordnung der Ereignisquellen auf eine höhere Parallelität zu skalieren. Weitere Informationen finden Sie unter Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen.
- Erhöhen Sie die Batchgröße der Funktion, sodass bei jedem Aufruf mehr Nachrichten verarbeitet werden.

Beheben von Netzwerkproblemen in Lambda

Standardmäßig führt Lambda Ihre Funktionen in einer internen Virtual Private Cloud (VPC) mit Konnektivität zu AWS -Services und dem Internet aus. Um auf lokale Netzwerkressourcen zuzugreifen, können Sie Ihre Funktion so konfigurieren, dass sie sich mit einer VPC in Ihrem Konto verbindet. Wenn Sie diese Funktion verwenden, verwalten Sie den Internetzugang und die Netzwerkkonnektivität der Lambda Event Function mit Amazon Virtual Private Cloud (Amazon VPC) -Ressourcen.

Netzwerkverbindungsfehler können auf Probleme mit der Routing-Konfiguration Ihrer VPC, den Sicherheitsgruppenregeln, AWS Identity and Access Management (IAM) -Rollenberechtigungen oder der Netzwerkadressübersetzung (NAT) oder auf die Verfügbarkeit von Ressourcen wie IP-Adressen oder Netzwerkschnittstellen zurückzuführen sein. Je nach Problem wird möglicherweise ein bestimmter Fehler oder Timeout angezeigt, wenn eine Anfrage ihr Ziel nicht erreichen kann.

Themen

- VPC: Funktion verliert Internetzugriff oder läuft ab
- VPC: TCP- oder UDP-Verbindung bricht zeitweise ab
- VPC: Funktion benötigt Zugriff auf AWS-Services ohne das Internet zu benutzen
- VPC: Grenzwert für Elastic-Network-Schnittstellen erreicht
- EC2: Elastische Netzwerkschnittstelle mit dem Typ „Lambda“
- DNS: Verbindung zu Hosts mit UNKNOWNHOSTEXCEPTION schlägt fehl

VPC: Funktion verliert Internetzugriff oder läuft ab

Problem: Ihre Lambda-Funktion verliert Internetzugriff nach der Verbindung mit einer VPC.

Fehler: Fehler: Verbindung ETIMEDOUT 176.32.98.189:443

Fehler: Fehler: Zeitüberschreitung der Aufgabe nach 10,00 Sekunden

FehlerReadTimeoutError: Timeout beim Lesen. (Timeout beim Lesen = 15)

Wenn Sie eine Funktion mit einer VPC verbinden, werden alle ausgehenden Anforderungen über die VPC geleitet. Um eine Verbindung zum Internet herzustellen, konfigurieren Sie Ihre VPC so, dass ausgehender Datenverkehr aus dem Subnetz der Funktion an ein NAT-Gateway in einem öffentlichen Subnetz gesendet wird. Weitere Informationen und VPC-Beispielkonfigurationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn einige Ihrer TCP-Verbindungen ein Timeout aufweisen, überprüfen Sie the section called “VPC: TCP- oder UDP-Verbindung bricht zeitweise ab”, ob Ihr Subnetz eine Netzwerkzugriffskontrollliste (NACL) verwendet. Andernfalls liegt dies wahrscheinlich an einer Paketfragmentierung. Lambda-Funktionen können eingehende fragmentierte TCP-Anfragen nicht verarbeiten, da Lambda keine IP-Fragmentierung für TCP oder ICMP unterstützt.

VPC: TCP- oder UDP-Verbindung bricht zeitweise ab

> **Note**
>
> Dieses Problem tritt nur auf, wenn Ihr Subnetz eine Zugriffssteuerungsliste (ACL) für Netzwerke verwendet. Netzwerk-ACLs sind nicht erforderlich, damit Lambda eine Verbindung zu Ihren Subnetzen herstellen kann.

Problem: Lambda verliert zeitweise die Verbindung zu Ihren VPC-Subnetzen, für die Sie eine Netzwerk-Zugriffssteuerungsliste (ACL) konfiguriert haben.

Für VPC-enabled Lambda-Funktionen AWS erstellt es Hyperplane-ENIs im Kundenkonto und verwendet kurzlebige Ports, um Lambda mit der VPC des Kunden `1024` `65535` zu verbinden. Wenn Sie Netzwerk-ACLs im Zielsubnetz verwenden, müssen Sie den Portbereich `1024` bis `65535` sowohl für TCP als auch für UDP zulassen. Wenn dieser vollständige Portbereich nicht zugelassen wird, kann es zu zeitweiligen Verbindungsausfällen kommen.

VPC: Funktion benötigt Zugriff auf AWS-Services ohne das Internet zu benutzen

Problem: Ihre Lambda-Funktion benötigt Zugriff auf, AWS-Services ohne das Internet zu verwenden.

Verwenden Sie VPC-Endpunkte, um eine Funktion AWS-Services von einem privaten Subnetz ohne Internetzugang aus zu verbinden.

VPC: Grenzwert für Elastic-Network-Schnittstellen erreicht

FehlerENILimitReachedException: Das elastic network interface Network-Schnittstellenlimit wurde für die VPC der Funktion erreicht.

Wenn Sie eine Funktion mit einer Lambda-VPC verbinden, erstellt Lambda für jede Kombination aus Subnetz und Sicherheitsgruppe, die der Funktion zugeordnet ist, eine Elastic-Network-Schnittstelle. Das Standard-Servicekontingent beträgt 3.000 Netzwerkschnittstellen pro VPC. Zum Anfordern einer Erhöhung für ein Kontingent können Sie die Service-Quotas-Konsole verwenden.

EC2: Elastische Netzwerkschnittstelle mit dem Typ „Lambda“

Fehlercode: Client.OperationNotPermitted

Fehlermeldung: Die Sicherheitsgruppe kann für diesen Schnittstellentyp nicht geändert werden

Sie erhalten diesen Fehler, wenn Sie versuchen, ein elastic network interface (ENI) zu ändern, das von Lambda verwaltet wird. Das `ModifyNetworkInterfaceAttribute` ist nicht in der Lambda-API für Aktualisierungsvorgänge auf elastischen Netzwerkschnittstellen enthalten, die von Lambda erstellt wurden.

DNS: Verbindung zu Hosts mit UNKNOWNHOSTEXCEPTION schlägt fehl

Fehlermeldung: UNKNOWNHOSTEXCEPTION

Lambda-Funktionen unterstützen maximal 20 gleichzeitige TCP-Verbindungen für die DNS-Auflösung. Ihre Funktion könnte dieses Limit ausgeschöpft haben. Die meisten DNS-Anfragen werden über UDP gestellt. Wenn Ihre Funktion nur UDP-DNS-Verbindungen herstellt, ist es unwahrscheinlich, dass dies Ihr Problem ist. Bevor Sie also Ihren DNS-Verkehr eingehend untersuchen, vergewissern Sie sich, dass Ihre DNS-Infrastruktur ordnungsgemäß konfiguriert und in Ordnung ist und dass sich Ihre Lambda-Funktion auf einen in DNS angegebenen Host bezieht.

Wenn Sie Ihr Problem im Zusammenhang mit der maximalen TCP-Verbindung diagnostizieren, beachten Sie, dass Sie keine Erhöhung dieses Limits beantragen können. Wenn Ihre Lambda-

Funktion aufgrund großer DNS-Payloads auf TCP DNS zurückgreift, überprüfen Sie, ob Ihre Lösung Bibliotheken verwendet, die EDNS unterstützen. Weitere Informationen über EDNS finden Sie in der Norm RFC 6891. Wenn Ihre DNS-Payloads durchweg die maximale Größe von EDNS überschreiten, kann es sein, dass Ihre Lösung das TCP-DNS-Limit immer noch ausschöpft.

Bewährte Methoden für die Arbeit mit AWS Lambda Funktionen für Ereignisse

Im Folgenden finden Sie einige bewährte Methoden für die Verwendung vo AWS Lambda:

Themen

- Code der Ereignisfunktion
- Konfiguration der Event-Funktion
- Skalierbarkeit der Event-Funktion
- Metriken und Alarme
- Arbeiten mit Streams
- Bewährte Methoden für die Gewährleistung der Sicherheit

Code der Ereignisfunktion

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung

aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Verbindungen mit Keep-Alive in Node.js wiederverwenden.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates auf die internen APIs von Lambda durch. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

> **Note**
>
> Sie können Powertools verwenden, um Funktionen idempotent AWS Lambda zu machen. Weitere Informationen finden Sie unter:
>
> - Python – Dienstprogramm für Idempotenz
> - TypeScript - Hilfsprogramm für Idempotenz
> - Java – Dienstprogramm für Idempotenz
> - .NET – Dienstprogramm für Idempotenz

Bewährte Methoden für sprachspezifischen Code finden Sie in den folgenden Abschnitten:

- Bewährte Codemethoden für Node.js Lambda-Ereignisfunktionen
- Bewährte Codemethoden für TypeScript Lambda-Ereignisfunktionen
- Bewährte Codemethoden für Python-Lambda-Ereignisfunktionen
- Bewährte Codemethoden für Ruby Lambda Event Functions
- Bewährte Codemethoden für Java Lambda Event Functions
- Bewährte Codemethoden für Go Lambda Event Functions
- Bewährte Codemethoden für Lambda-Ereignisfunktionen in C#
- Bewährte Codemethoden für Rust Lambda Event Functions

Konfiguration der Event-Funktion

Der Leistungstest Ihrer Lambda Event Function ist entscheidend, um sicherzustellen, dass Sie die optimale Speichergrößenkonfiguration auswählen. Jede Erhöhung des Arbeitsspeichers führt zu einer gleichwertigen Erhöhung der CPU-Verfügbarkeit Ihrer Funktion. Der Speicherverbrauch für Ihre Funktion wird pro Aufruf bestimmt und kann in Amazon eingesehen werden. CloudWatch Bei jedem Aufruf wird ein `REPORT:` Eintrag vorgenommen, wie unten gezeigt:

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
 Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

Durch die Analyse `Max Memory Used:` des Feldes können Sie feststellen, ob Ihre Funktion mehr Speicher benötigt oder ob die Speichergröße Ihrer Funktion überdimensioniert ist.

Um die richtige Speicherkonfiguration für Ihre Funktionen zu finden, empfehlen wir die Verwendung des Open-Source-Projekts AWS Lambda Power Tuning. Weitere Informationen finden Sie unter AWS Lambda Power Tuning on GitHub.

Um die Funktionsleistung zu optimieren, empfehlen wir außerdem den Einsatz von Bibliotheken, die Advanced Vector Extensions 2 (AVX2) verwenden können. Damit können Sie anspruchsvolle Workloads verarbeiten, darunter maschinelles Lernen, Inferenz, Medienverarbeitung, Hochleistungsrechnen (HPC), wissenschaftliche Simulationen und Finanzmodellierung. Weitere Informationen finden Sie unter Schnellere AWS Lambda Funktionen mit AVX2 erstellen.

Testen Sie Ihre Lambda Event Function unter Last, um einen optimalen Timeout-Wert zu ermitteln. Es ist wichtig zu analysieren, wie lange Ihre Funktion ausgeführt wird, damit Sie Probleme mit einem Abhängigkeitsdienst, die die Parallelität der Funktion über das erwartete Maß hinaus erhöhen könnten, besser erkennen können. Dies ist besonders wichtig, wenn Ihre Lambda-Ereignisfunktion

Netzwerkaufrufe an Ressourcen durchführt, die die Skalierung von Lambda möglicherweise nicht bewältigen können. Weitere Informationen über Lasttests Ihrer Anwendung finden Sie unter Verteilte Lasttests auf AWS.

Verwenden Sie die strengsten Berechtigungen bei der Einrichtung von IAM-Richtlinien. Machen Sie sich mit den Ressourcen und Vorgängen vertraut, die Ihre Lambda Event Function benötigt, und beschränken Sie die Ausführungsrolle auf diese Berechtigungen. Weitere Informationen finden Sie unter Verwaltung von Berechtigungen in AWS Lambda.

Machen Sie sich mit Lambda-Kontingente vertraut. Nutzlastgröße, Dateideskriptoren und /tmp space werden bei der Ermittlung von Laufzeit-Ressourcen-Limits oft übersehen.

Löschen Sie Lambda-Ereignisfunktionen, die Sie nicht mehr verwenden. Auf diese Weise werden die nicht ungenutzten Funktionen nicht unnötigerweise auf die Größenbeschränkung Ihres Bereitstellungspakets angerechnet.

Wenn Sie Amazon Simple Queue Service als Ereignisquelle verwenden, stellen Sie sicher, dass der Wert der erwarteten Aufrufzeit der Funktion nicht größer ist als der Wert der Zeitbeschränkung für die Sichtbarkeit in der Warteschlange. Dies gilt für CreateFunction und für UpdateFunctionConfiguration.

- Im Fall von AWS Lambda schlägt CreateFunctiondie Funktionserstellung fehl.
- Im Fall von UpdateFunctionConfigurationkönnte dies zu doppelten Aufrufen der Funktion führen.

Skalierbarkeit der Event-Funktion

Machen Sie sich mit Ihren vor- und nachgelagerten Durchsatzbeschränkungen vertraut. Lambda Event Functions lassen sich zwar nahtlos mit der Last skalieren, aber Upstream- und Downstream-Abhängigkeiten verfügen möglicherweise nicht über die gleichen Durchsatzfähigkeiten. Wenn Sie die Skalierbarkeit Ihrer Funktion begrenzen müssen, können Sie die reservierte Gleichzeitigkeit für Ihre Funktion konfigurieren.

Integrierte Drosseltoleranz. Wenn bei Ihrer synchronen Funktion eine Drosselung auftritt, weil der Datenverkehr die Skalierungsrate von Lambda überschreitet, können Sie die folgenden Strategien anwenden, um die Drosselungstoleranz zu verbessern:

- Timeouts, Wiederholungen und Backoff mit Jitter verwenden. Durch die Implementierung dieser Strategien werden wiederholte Aufrufe vereinfacht und es wird sichergestellt, dass Lambda innerhalb von Sekunden skaliert werden kann, um die Drosselung durch Endbenutzer zu minimieren.

- Bereitgestellte Parallelität verwenden. Die bereitgestellte Nebenläufigkeit beschreibt die Anzahl der vorinitialisierten Ausführungsumgebungen, die Lambda Ihrer Funktion zuweist. Lambda verarbeitet eingehende Anforderungen mit bereitgestellter Gleichzeitigkeit, sofern verfügbar. Lambda kann Ihre Funktion bei Bedarf auch über die vorgesehene Gleichzeitigkeitseinstellung hinaus skalieren. Durch die Konfiguration der bereitgestellten Parallelität fallen zusätzliche Gebühren für Ihr Konto an. AWS

Metriken und Alarme

Verwenden Sie CloudWatch Metriken mit Lambda verwenden und CloudWatch Alarms, anstatt eine Metrik in Ihrem Lambda Event Function-Code zu erstellen oder zu aktualisieren. Es ist eine viel effizientere Methode, den Zustand Ihrer Lambda Event Functions zu verfolgen, sodass Sie Probleme catch im Entwicklungsprozess erkennen können. Sie können beispielsweise einen Alarm auf der Grundlage der erwarteten Dauer Ihres Aufrufs der Lambda-Ereignisfunktion konfigurieren, um Engpässe oder Latenzen zu beheben, die auf Ihren Funktionscode zurückzuführen sind.

Geben Sie benutzerdefinierte Metriken asynchron mithilfe des Embedded Metric Format (EMF) aus. Anstatt synchrone API-Aufrufe zu tätigen CloudWatch, verwenden Sie EMF, um Metriken über die Protokolle Ihrer Funktion auszugeben. Dieser Ansatz reduziert die Latenz und verbessert die Leistung.

Das Metrics-Dienstprogramm in Powertools for AWS Lambda verarbeitet die EMF-Formatierung automatisch. Weitere Informationen finden Sie in der AWS Lambda Dokumentation unter Python - TypeScript, Java - oder .NET Metrics-Dienstprogramme in den Powertools. Informationen zur Verwendung von EMF zur Generierung von Protokollen im Metrikformat finden Sie unter Veröffentlichen von Protokollen mit dem eingebetteten metrischen Format im CloudWatch Amazon-Benutzerhandbuch.

Verwenden Sie die strukturierte JSON-Protokollierung für eine bessere Beobachtbarkeit. Die strukturierte Protokollierung erleichtert das Suchen, Filtern und Analysieren der Protokolle Ihrer Funktion. Erwägen Sie die Verwendung des Logger-Dienstprogramms von Powertools für die AWS Lambda automatische Formatierung von Protokollen in JSON. Weitere Informationen finden Sie in der AWS Lambda Dokumentation unter Python - TypeScript, Java - oder .NET Logger-Dienstprogramme in den Powertools.

Verwenden Sie Ihre Logging-Bibliothek und AWS Lambda Metriken und Dimensionen catch um App-Fehler (wie ERR, ERROR und WARNING) zu erkennen.

Verwenden Sie die AWS -Kostenanomalie-Erkennung, um ungewöhnliche Aktivitäten in Ihrem Konto zu erkennen. Die Kostenanomalie-Erkennung verwendet Machine Learning, um Ihre Kosten und Ihre Nutzung kontinuierlich zu überwachen und gleichzeitig falsch positive Warnungen zu minimieren. Die Erkennung von Kostenanomalien verwendet Daten von AWS Cost Explorer, was zu einer Verzögerung von bis zu 24 Stunden führt.

Daher kann es nach der Nutzung bis zu 24 Stunden dauern, bis eine Anomalie erkannt wird. Um die Kostenanomalie-Erkennung verwenden zu können, müssen Sie sich zunächst bei Cost Explorer anmelden. Anschließend können Sie auf die Kostenanomalie-Erkennung zugreifen.

Arbeiten mit Streams

Testen Sie mit unterschiedlichen Stapel- und Datensatzgrößen, so dass die Abfragefrequenz jeder Ereignisquelle darauf abgestimmt ist, wie schnell Ihre Funktion ihre Aufgabe erledigen kann. Der CreateEventSourceMapping BatchSize Parameter steuert die maximale Anzahl von Datensätzen, die bei jedem Aufruf an Ihre Funktion gesendet werden können. Eine höhere Stapelgröße kann den mit dem Aufruf-Overhead über eine größere Datensatzgruppe hinweg oft effizienter verarbeiten und Ihren Durchsatz erhöhen.

Standardmäßig ruft Lambda Ihre Funktion auf, sobald Datensätze verfügbar sind. Wenn der Batch, den Lambda aus der Ereignisquelle liest, nur einen Datensatz enthält, sendet Lambda nur einen Datensatz an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle anweisen, Datensätze bis zu 5 Minuten lang zu puffern, indem Sie ein Batch-Fenster konfigurieren. Bevor die Funktion aufgerufen wird, liest Lambda so lange Datensätze aus der Ereignisquelle, bis es einen vollständigen Batch erfasst hat, das Batch-Verarbeitungsfenster abläuft oder der Batch die Nutzlastgrenze von 6 MB erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Aktivieren Sie die teilweise Batch-Antwort für die Stream-Verarbeitung. Wenn Sie Batches von Datensätzen aus Streams wie Kinesis oder DynamoDB Streams verarbeiten, aktivieren Sie die

teilweise Batch-Antwort, damit Lambda nur die fehlgeschlagenen Datensätze und nicht den gesamten Batch erneut versuchen kann. Dies verbessert die Verarbeitungseffizienz und reduziert unnötige Wiederholungsvorgänge. Sie können optional das Batch-Hilfsprogramm von Powertools verwenden AWS Lambda , um die Stapelverarbeitungsmuster zu vereinfachen.

> **Note**
>
> Sie können Powertools AWS Lambda für die Stapelverarbeitung verwenden. Weitere Informationen finden Sie unter:
>
> - Python – Batchverarbeitung
> - TypeScript - Stapelverarbeitung
> - Java – Batchverarbeitung
> - .NET – Batchverarbeitung

Erhöhen Sie den Durchsatz bei der Verarbeitung des Kinesis-Streams durch Hinzufügen von Shards. Ein Kinesis Stream besteht aus einem oder mehreren Shards. Die Geschwindigkeit, mit der Lambda Daten von Kinesis lesen kann, skaliert linear mit der Anzahl der Shards.

Eine direkte Erhöhung der Anzahl der Shards erhöht die Anzahl der maximalen gleichzeitigen Aufrufe der Lambda-Ereignisfunktion und kann den Durchsatz der Kinesis-Stream-Verarbeitung erhöhen. Weitere Informationen über die Beziehung zwischen Shards und Funktionsaufrufen finden Sie unter the section called “Abfragen und Stapeln von Streams”. Wenn Sie die Anzahl der Shards in einem Kineses-Stream erhöhen, stellen Sie sicher, dass Sie für Ihre Daten einen guten Partitionsschlüssel ausgewählt haben, (siehe Partitionsschlüssel), so dass Bezugsdatensätze auf denselben Shards gespeichert werden und Ihre Daten gut verteilt sind.

Verwenden Sie Amazon CloudWatch on IteratorAge , um festzustellen, ob Ihr Kinesis-Stream verarbeitet wird. Konfigurieren Sie beispielsweise einen CloudWatch Alarm mit einer Maximaleinstellung von 30000 (30 Sekunden).

Bewährte Methoden für die Gewährleistung der Sicherheit

Überwachen Sie Ihre Nutzung von AWS Lambda in Bezug auf bewährte Sicherheitsmethoden mithilfe AWS Security Hub CSPM von. Security Hub CSPM verwendet Sicherheitskontrollen, um Ressourcenkonfigurationen und Sicherheitsstandards zu bewerten und Sie bei der Einhaltung verschiedener Compliance-Frameworks zu unterstützen. Weitere Informationen zur Verwendung

von Security Hub CSPM zur Evaluierung von Lambda-Ressourcen finden Sie unter AWS Lambda Kontrollen im AWS Security Hub CSPM Benutzerhandbuch.

Überwachen Sie Lambda-Netzwerkaktivitätsprotokolle mit Amazon GuardDuty Lambda Protection. GuardDuty Lambda-Schutz hilft Ihnen dabei, potenzielle Sicherheitsbedrohungen zu identifizieren, wenn Lambda Event Functions in Ihrem aufgerufen werden. AWS-Konto Dies ist beispielsweise der Fall, wenn eine Ihrer Funktionen eine IP-Adresse abfragt, die mit Aktivitäten im Zusammenhang mit Kryptowährungen verknüpft ist. GuardDuty überwacht die Netzwerkaktivitätsprotokolle, die generiert werden, wenn eine Lambda-Ereignisfunktion aufgerufen wird. Weitere Informationen finden Sie unter Lambda-Schutz im GuardDuty Amazon-Benutzerhandbuch.

Lambda-Beispielanwendungen

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung verschiedener Sprachen und AWS Dienste demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung und unterstützende Ressourcen.

Node.js

Beispiele für Lambda-Anwendungen in Node.js

- blank-nodejs — Eine Node.js Funktion, die die Verwendung von Logging, Umgebungsvariablen, AWS X-Ray Tracing, Layern, Unit-Tests und dem SDK zeigt. AWS
- nodejs-apig – Eine Funktion mit einem öffentlichen API-Endpunkt, die ein Ereignis aus API Gateway verarbeitet und eine HTTP-Antwort zurückgibt.

Python

Lambda-Beispielanwendungen in Python

- blank-python — Eine Python-Funktion, die die Verwendung von Logging, Umgebungsvariablen, AWS X-Ray Tracing, Layern, Unit-Tests und dem SDK zeigt. AWS

Ruby

Lambda-Beispielanwendungen in Ruby

- blank-ruby — Eine Ruby-Funktion, die die Verwendung von Logging, Umgebungsvariablen, AWS X-Ray Tracing, Layern, Unit-Tests und dem SDK zeigt. AWS

- Ruby-Codebeispiele für AWS Lambda — In Ruby geschriebene Codebeispiele, die zeigen, wie man mit AWS Lambda interagiert.

Java

Lambda-Beispielanwendungen in Java

- example-java — Eine Java-Event-Funktion, die demonstriert, wie Sie Lambda zur Bearbeitung von Bestellungen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das SDK verwenden und wie Sie die AWS Ausgabe protokollieren.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Ausführen beliebter Java-Frameworks auf Lambda

- spring-cloud-function-samples — Ein Beispiel aus Spring, das zeigt, wie das Spring Cloud Function Framework verwendet wird, um Lambda Event Functions zu erstellen. AWS
- Serverlose Spring Boot-Anwendungsdemo — Ein Beispiel, das zeigt, wie eine typische Spring Boot-Anwendung in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit eingerichtet wird.
- Serverlose Micronaut-Anwendungsdemo — Ein Beispiel, das zeigt, wie Micronaut in einer verwalteten Java-Laufzeit mit und ohne oder als natives SnapStart GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet wird. Erfahren Sie mehr in den Leitfäden. Micronaut/Lambda

- Serverlose Quarkus-Anwendungsdemo — Ein Beispiel, das zeigt, wie Quarkus in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet wird. Weitere Informationen finden Sie in der Anleitung und im Leitfaden. Quarkus/Lambda Quarkus/SnapStart

Go

Lambda stellt die folgenden Beispielanwendungen für die Go-Laufzeit bereit:

Lambda-Beispielanwendungen in Go

- go-al2: Eine Hello World-Funktion, die die öffentliche IP-Adresse zurückgibt. Diese App verwendet die benutzerdefinierte Laufzeit `provided.al2`.
- blank-go — Eine Go-Funktion, die die Verwendung der Go-Bibliotheken, der Protokollierung, der Umgebungsvariablen und des SDK von Lambda zeigt. AWS Diese App verwendet die Laufzeit `go1.x`.

C#

Lambda-Beispielanwendungen in C#

- blank-csharp AWS X-Ray Eine C #-Funktion, welche die Verwendung der Lambda-NET-Bibliotheken, Protokollierung, Umgebungsvariablen, -Nachverfolgung, Einheitentests und des AWS SDK aufzeigt.
- blank-csharp-with-layer – Eine C#-Funktion, die die .NET-CLI verwendet, um eine Ebene zu erstellen, die die Abhängigkeiten der Funktion bündelt.
- ec2-spot – Eine Funktion, die Spot-Instance-Anforderungen in Amazon EC2 verwaltet.

PowerShell

Lambda bietet die folgenden Beispielanwendungen für PowerShell:

- blank-powershell — Eine PowerShell Funktion, die die Verwendung von Logging, Umgebungsvariablen und dem SDK veranschaulicht. AWS

Um eine Beispielanwendung bereitzustellen, befolgen Sie die Anweisungen in der README-Datei.

Entwicklungstools für Lambda

Sie haben Zugriff auf eine Vielzahl von Tools, die die Produktivität und Benutzerfreundlichkeit während des gesamten Entwicklungszyklus steigern. Dieser Abschnitt enthält Informationen zu Tools, die vielen Lambda-Kunden beim Entwerfen, Entwickeln und Verwalten ihrer Anwendungen helfen. Von der lokalen Entwicklung in Ihrer IDE bis hin zur Bereitstellung und Verwaltung komplexer serverloser Anwendungen helfen Ihnen diese Tools dabei, Ihren Arbeitsablauf zu optimieren, die Codequalität zu verbessern und die Entwicklung robuster Lambda-based Lösungen zu beschleunigen.

- Lokale Entwicklung — Schreiben und testen Sie Lambda Event Functions schneller in Ihrer bevorzugten Entwicklungsumgebung. Das AWS Toolkit for VS Code ermöglicht die lokale Funktionsentwicklung, das Debuggen und Testen mit direkten Bereitstellungsfunktionen für Lambda.
- Infrastructure as Code (IaC) — Stellen Sie serverlose Anwendungen vom lokalen Testen bis zur Produktion konsistent bereit und verwalten Sie sie. AWS SAM AWS CDK, und CloudFormation ermöglicht es Ihnen, Ihre serverlose Infrastruktur mithilfe von Code für konsistente, versionskontrollierte Bereitstellungen zu definieren und zu verwalten.
- GitHub Aktionen — Automatisieren Sie Lambda-Bereitstellungen direkt aus Ihrem Code-Repository. Mit GitHub Aktionen können Sie Workflows einrichten, die Ihre Lambda-Funktionen automatisch bereitstellen, wenn Sie Code- oder Konfigurationsänderungen veröffentlichen, wodurch Ihre CI/CD Pipeline vereinfacht wird.
- Powertools for AWS Lambda — Erstellen Sie produktionsreife serverlose Anwendungen mit weniger benutzerdefiniertem Code. Powertools for AWS Lambda (auch als Powertools for bezeichnet AWS) ist ein Open-Source-Entwickler-Toolkit, das die Implementierung serverloser Best Practices wie Observability, Parameterabruf und Idempotenz in Python, Java und .NET vereinfacht. TypeScript
- Workflows und Ereignisse — Koordinieren Sie Lambda-Funktionen mit AWS Diensten, APIs und externen Systemen. Lambda bietet zwei Orchestrierungsoptionen: langlebige Lambda-Funktionen für die anwendungsorientierte Orchestrierung mithilfe von Standardprogrammiersprachen innerhalb von Lambda und für die workflow-zentrierte Orchestrierung mit visuellem AWS Step FunctionsDesign für mehrere Dienste. Amazon EventBridge bietet Event-Management-Funktionen für ereignisgesteuerte Architekturen. Hilfe bei der Auswahl eines Orchestrierungsansatzes finden Sie unter Durable functions oder Step Functions.

Lokale Entwicklungstools

Lokale Entwicklungsumgebungen ermöglichen es Ihnen, offline zu arbeiten und erweiterte IDE-Funktionen zu verwenden, während Sie schnell an Ihren Lambda-Funktionen arbeiten. Diese Tools helfen Ihnen beim Debuggen komplexer Funktionen und bei der Entwicklung in Umgebungen mit eingeschränkter Konnektivität. Sie unterstützen auch die Zusammenarbeit im Team und die Integration mit Versionskontrollsystemen.

Weitere Informationen zum lokalen Entwickeln von Lambda-Funktionen finden Sie unter Lokale Entwicklung von Eventfunktionen. Auf dieser Seite wird beschrieben, wie Sie die Lambda-Funktionsentwicklung von der AWS Konsole auf Visual Studio Code übertragen, das eine umfangreiche Entwicklungsumgebung mit Funktionen wie Debugging und Codevervollständigung bietet. Für den Übergang müssen Sie die Anmeldeinformationen AWS Toolkit for Visual Studio Code und die Anmeldeinformationen einrichten. Anschließend können Sie erweiterte Funktionen in VS Code verwenden und gleichzeitig die Möglichkeit zur direkten Bereitstellung beibehalten. AWS

Die lokale Entwicklung für Lambda bietet mehrere wichtige Funktionen:

- Verwenden der Visual-Studio-Code-Integration mit der Lambda-Konsole
- Konfigurieren lokaler Lambda-Entwicklungsumgebungen
- Lokales Debuggen und Testen von Funktionen
- Anwenden von Best Practices für lokale Funktionsverwaltung

Weitere Informationen finden Sie unter Lokale Entwicklung von Eventfunktionen.

Infrastructure as Code (IaC)-Tools

Mit den Tools für Infrastructure as Code (IaC) können Sie Ihre Serverless-Architektur mithilfe von Code definieren und verwalten. Dieser Ansatz trägt dazu bei, die Konsistenz in allen Umgebungen aufrechtzuerhalten, gibt Ihnen die Kontrolle über Ihre Infrastrukturversionen und unterstützt DevOps Verfahren. IaC ist besonders wertvoll für die Automatisierung von Bereitstellungen, die Gewährleistung konsistenter Umgebungen und die Verwaltung von Bereitstellungen in mehreren Regionen.

Zu den wichtigsten IaC-Tools und -Konzepten für Lambda gehören Frameworks für die Erstellung von Vorlagen, das Bereitstellungsmanagement und Best Practices für Serverless-Infrastrukturen:

- Kernprinzipien von IaC für die Lambda-Entwicklung

- CloudFormation AWS SAM, und AWS CDK Funktionen
- Auswahlkriterien und Vergleich von Tools
- Best Practices für die Implementierung von Lambda IaC

Ganz gleich, ob Sie selbstständig an einem kleinen Projekt arbeiten oder als Teil eines großen Teams unternehmensweite Serverless-Anwendungen verwalten – diese Entwicklungs- und Bereitstellungstools helfen Ihnen dabei, Ihre Lambda-Funktionen effektiver zu schreiben, bereitzustellen und zu verwalten.

Weitere Informationen finden Sie unter Verwenden von Lambda mit Infrastructure as Code (IaC).

GitHub Aktionen, Tools

GitHub Actions bietet automatisierte Bereitstellungsfunktionen für Ihre Lambda-Funktionen direkt aus Ihrem Code-Repository. Durch die Erstellung von Workflow-Dateien in Ihrem Repository können Sie Lambda-Funktionen automatisch bereitstellen, wann immer Code- oder Konfigurationsänderungen übertragen werden, wodurch Ihre kontinuierliche Integration und Continuous Deployment (CI/CD) -Pipeline optimiert wird. Die Funktionsaktion Deploy Lambda bietet eine deklarative YAML-Schnittstelle, die den Bereitstellungsprozess vereinfacht, AWS Anmeldeinformationen über OpenID Connect (OIDC) verarbeitet und verschiedene Bereitstellungsszenarien unterstützt, darunter Codeaktualisierungen, Konfigurationsänderungen und Testlaufvalidierungen. Diese Integration ermöglicht es Teams, einen konsistenten und automatisierten Bereitstellungsprozess aufrechtzuerhalten und gleichzeitig ihre bestehenden Workflows zu nutzen. GitHub

Weitere Informationen finden Sie unter Verwenden von GitHub Aktionen zum Bereitstellen von Ereignisfunktionen.

Powertools für AWS Lambda

Powertools for AWS ist ein Open-Source-Entwickler-Toolkit, mit dem Sie serverlose Best Practices mit minimalem benutzerdefiniertem Code implementieren können. Es ist für Python TypeScript/Node.js, Java und .NET verfügbar und bietet Hilfsfunktionen, Dekoratoren und Middleware, die allgemeine Lambda-Entwicklungsaufgaben rationalisieren. Das Toolkit umfasst integrierte Funktionen für die Beobachtbarkeit wie strukturierte Protokollierung, Tracing und Metrikenerfassung, darunter Dienstprogramme für die Parameterabfrage, die Verwaltung geheimer Schlüssel und Idempotenzmuster. Diese Tools orientieren sich an AWS gut durchdachten Best Practices und helfen Entwicklern dabei, produktionsreife serverlose Anwendungen effizienter zu erstellen. Durch

die Reduzierung von Standardcode und die Standardisierung gängiger Muster AWS ermöglicht Powertools for Teams, sich auf die Geschäftslogik zu konzentrieren und gleichzeitig die konsistente Implementierung serverloser Best Practices in ihren Anwendungen aufrechtzuerhalten.

Weitere Informationen finden Sie unter Elektrowerkzeuge für AWS Lambda.

Tools für die Workflow- und Ereignisverwaltung

Lambda-Anwendungen können zur Orchestrierung komplexer Workflows und zur Behandlung verschiedener Ereignisse verwendet werden. AWS bietet spezielle Tools, mit denen Sie diese Aspekte der serverlosen Entwicklung verwalten können. Erfahren Sie mehr über AWS Step Functions Workflow-Orchestrierung und Amazon EventBridge für Eventmanagement und wie Sie diese in Ihre Lambda-Funktionen integrieren können. Diese Tools können die Skalierbarkeit und Zuverlässigkeit Ihrer Serverless-Anwendungen erheblich verbessern, indem sie eine robuste Zustandsverwaltung und ereignisgesteuerte Architekturen bieten. Mithilfe dieser Services können Sie ausgefeiltere und robustere Lambda-based Lösungen entwickeln, die komplexe Geschäftsprozesse bewältigen und auf eine Vielzahl von System- und Anwendungsereignissen reagieren können.

Weitere Informationen finden Sie unter Verwaltung von Lambda-Workflows und -Ereignissen.

Lokale Entwicklung von Eventfunktionen

Sie können Ihre Lambda-Ereignisfunktionen von der Lambda-Konsole in Ihre bevorzugte integrierte Entwicklungsumgebung (IDE) verschieben, die eine vollständige Entwicklungsumgebung bietet, in der Sie andere lokale Entwicklungsoptionen wie AWS SAM und verwenden können. AWS CDK Die Lambda-Konsole unterstützt das direkte Öffnen von Funktionen in Visual Studio Code, Kiro oder Cursor.

Hauptvorteile der lokalen Entwicklung

Während die Lambda-Konsole eine schnelle Möglichkeit zum Bearbeiten und Testen von Funktionen bietet, verfügt die lokale Entwicklung über erweiterte Funktionen:

- Erweiterte IDE-Features: Tools für Debugging, Codevervollständigung und Faktorwechsel
- Offline-Entwicklung: lokales Arbeiten und Testen von Änderungen vor der Cloud-Bereitstellung
- Integration von Infrastruktur als Code: Nahtlose Verwendung mit AWS SAM AWS CDK, und Infrastructure Composer
- Abhängigkeitsmanagement: vollständige Kontrolle über Funktionsabhängigkeiten

Voraussetzungen

Bevor Sie Lambda-Funktionen lokal mit Visual Studio Code, Kiro und Cursor entwickeln, müssen Sie sie mithilfe der folgenden Anweisungen installieren:

- Visual Studio Code: Installationsanweisungen finden Sie unter Visual Studio Code herunterladen.
- Kiro: Installationsanweisungen finden Sie unter Erste Schritte mit Kiro.
- Cursor: Installationsanweisungen finden Sie unter Cursor herunterladen.

Sie benötigen außerdem Folgendes:

- AWS Toolkit for Visual Studio Code: Anweisungen zur Installation von Visual Studio Code finden Sie unter Einrichten von AWS Toolkit for Visual Studio Code. Eine Übersicht finden Sie unter AWS Toolkit for Visual Studio Code.
- AWS Anmeldeinformationen: Informationen zur Konfiguration von Anmeldeinformationen finden Sie unter AWS Anmeldeinformationen einrichten.
- AWS SAMCLI: Installationsanweisungen finden Sie unter Installation von AWS SAMCLI.
- Docker installiert (optional, aber für lokale Tests erforderlich): Eine Anleitung zum Installieren finden Sie unter Docker erhalten.

> **Note**
>
> Wenn Sie bereits ein AWS Konto und ein Profil lokal konfiguriert haben, stellen Sie sicher, dass die AdministratorAccess verwaltete Richtlinie zu Ihrem konfigurierten AWS Profil hinzugefügt wurde.

Authentifizierung und Zugriffskontrolle

Um Lambda-Funktionen lokal zu entwickeln, benötigen Sie AWS Anmeldeinformationen, um in Ihrem Namen sicher auf AWS Ressourcen zuzugreifen und diese zu verwalten, genau wie in der Cloud.

Das AWS Toolkit for Visual Studio Code unterstützt die folgenden Authentifizierungsmethoden:

- Langfristige Anmeldeinformationen für IAM-Benutzer
- Temporäre Anmeldeinformationen von angenommenen Rollen

- Identitätsverbund
- AWS Root-Benutzeranmeldedaten für das Konto (nicht empfohlen)

Dieser Abschnitt führt Sie durch den Prozess zum Abrufen und Konfigurieren dieser Anmeldeinformationen mithilfe der langfristigen Anmeldeinformationen für IAM-Benutzer.

Erhalt von IAM-Anmeldeinformationen

Sie haben bereits einen IAM-Benutzer mit Zugriffsschlüsseln? Halten Sie die Zugriffsschlüssel-ID und den geheimen Zugriffsschlüssel für den nächsten Abschnitt bereit. Wenn nicht, erstellen Sie diese Elemente wie folgt:

> **Note**
>
> Sie müssen die Zugriffsschlüssel-ID und den geheimen Zugriffsschlüssel zusammen verwenden, um Ihre Anforderungen zu authentifizieren.

So erstellen Sie einen IAM-Benutzer und Zugriffsschlüssel:

1. Öffnen Sie die IAM-Konsole unter https://console.aws.amazon.com/iam/
2. Klicken Sie im Navigationsbereich auf Users (Benutzer).
3. Wählen Sie Create user (Benutzer erstellen) aus.
4. Geben Sie unter Benutzername einen Namen ein und wählen Sie Weiter.
5. Wählen Sie unter Berechtigungen festlegen die Option Richtlinien direkt anhängen.
6. Wählen AdministratorAccessund wählen Sie Weiter.
7. Wählen Sie Create user (Benutzer erstellen) aus.
8. Wählen Sie im Banner mit der Erfolgsmeldung die Option Benutzer anzeigen.
9. Wählen Sie Zugriffsschlüssel erstellen aus.
10. Wählen Sie für Anwendungsfall die Option Lokaler Code.
11. Aktivieren Sie das Kontrollkästchen für die Bestätigung und wählen Sie Weiter.
12. (Optional) Geben Sie einen Wert für das Beschreibungs-Tag ein.
13. Wählen Sie Zugriffsschlüssel erstellen aus.
14. Kopieren Sie sofort Ihren Zugriffsschlüssel und den geheimen Zugriffsschlüssel. Sie können nach dem Verlassen dieser Seite nicht mehr auf den geheimen Zugriffsschlüssel zugreifen.

> **Important**
>
> Geben Sie Ihren geheimen Schlüssel niemals weiter und übertragen Sie ihn nicht in die Quellcodeverwaltung. Bewahren Sie diese Schlüssel sicher auf und löschen Sie sie, wenn sie nicht mehr benötigt werden.

> **Note**
>
> Weitere Informationen finden Sie unter Einen IAM-Benutzer in Ihrem AWS Konto erstellen und Zugriffsschlüssel für IAM-Benutzer verwalten im IAM-Benutzerhandbuch.

Konfiguration AWS Anmeldeinformationen mit dem AWS Toolkit

In der folgenden Tabelle wird der Vorgang zur Einrichtung der Anmeldeinformationen zusammengefasst, den Sie im folgenden Verfahren durchführen.

Vorgehensweise	Warum?
Öffnen Sie das Anmeldefenster.	Starten der Authentifizierung
Suchen Sie in der Befehlspalette nach AWS Neue Verbindung hinzufügen	Greifen Sie auf die Anmeldebenutzeroberfläche zu.
Wählen Sie die IAM-Anmeldeinformationen aus.	Verwenden Sie Ihren Zugriffsschlüssel für den programmatischen Zugriff.
Geben Sie den Profilnamen, den Zugriffss chlüssel und den geheimen Schlüssel ein.	Geben Sie die Anmeldeinformationen für die Verbindung ein.
Siehe AWS Explorer-Update	Bestätigen Sie, dass Sie verbunden sind.

Gehen Sie wie folgt vor, um sich bei Ihrem AWS Konto zu authentifizieren:

> **Note**
>
> Die folgenden Schritte gelten für Visual Studio Code, Kiro und Cursor. Da Kiro und Cursor auf Visual Studio Code basieren, ist der Authentifizierungsablauf in allen drei IDEs derselbe.

1. Öffnen Sie das Anmeldefenster in Ihrer IDE:

   Um den Authentifizierungsprozess zu starten, wählen Sie das AWS Symbol im linken Navigationsbereich oder öffnen Sie die Befehlspalette (Cmd+Shift+P auf dem Mac oder Strg+Shift +P auf Windows/Linux) und suchen Sie nach Neue Verbindung hinzufügen und wählen Sie diese aus.AWS
2. Wählen Sie im Anmeldebereich IAM-Anmeldeinformationen und anschließend Weiter.

   > **Note**
   >
   > Um fortzufahren, müssen Sie IDE-Erweiterungen den Zugriff auf Ihre Daten erlauben. AWS
3. Geben Sie Ihren Profilnamen, Ihre Zugriffsschlüssel-ID und Ihren geheimen Zugriffsschlüssel ein und wählen Sie dann Weiter aus.
4. Überprüfen Sie die Verbindung, indem Sie im AWS Explorer in Ihrer IDE nach Ihren AWS Diensten und Ressourcen suchen.

Informationen zum Einrichten der Authentifizierung mit langfristigen Anmeldeinformationen finden Sie unter Verwenden langfristiger Anmeldeinformationen zur Authentifizierung von AWS SDKs und Tools.

Informationen zur Konfiguration der Authentifizierung finden Sie unter AWS IAM-Anmeldeinformationen im AWS Toolkit for Visual Studio Code Benutzerhandbuch.

Wechsel von der Konsole zur lokalen Entwicklung

> **Note**
>
> Wenn Sie Änderungen in der Konsole vorgenommen haben, stellen Sie sicher, dass keine nicht bereitgestellten Änderungen vorhanden sind, bevor Sie zur lokalen Entwicklung übergehen.

Gehen Sie wie folgt vor, um eine Lambda-Funktion von der Lambda-Konsole in eine lokale IDE zu verschieben:

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie die Registerkarte Codequelle aus.
4. Öffnen Sie Ihre Funktion in der IDE Ihrer Wahl, indem Sie den Dropdown-Pfeil neben der Schaltfläche auswählen. Die unterstützten Optionen sind In Visual Studio Code öffnen, In Kiro öffnen und In Cursor öffnen.

   > **Note**
   >
   > Ihre IDE-Auswahl wird beibehalten. Sobald Sie eine IDE ausgewählt haben, wird bei nachfolgenden Besuchen diese IDE als primäre Schaltfläche angezeigt.

5. Wenn Sie dazu aufgefordert werden, erlauben Sie Ihrem Browser, die ausgewählte IDE zu öffnen.

Wenn Sie Ihre Funktion in Ihrer IDE öffnen, erstellt Lambda ein lokales Projekt mit Ihrem Funktionscode an einem temporären Speicherort, das für schnelles Testen und Bereitstellen konzipiert ist. Dazu gehören der Funktionscode, Abhängigkeiten und eine grundlegende Projektstruktur, die Sie für die lokale Entwicklung verwenden können.

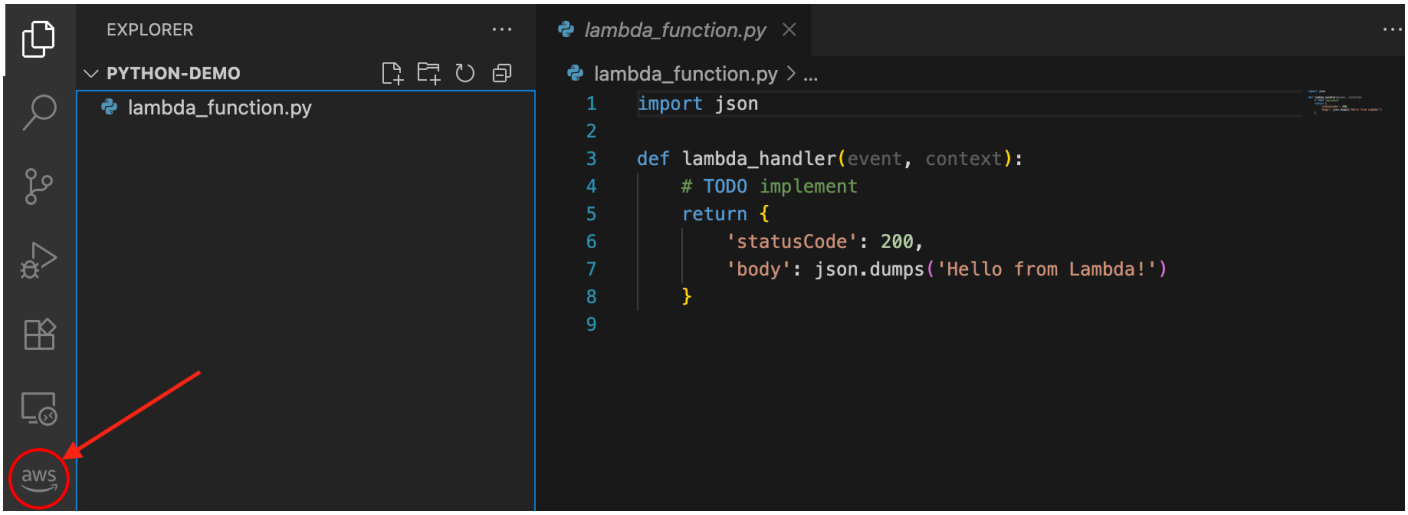
> **Note**
>
> Für die Schaltfläche In IDE öffnen ist die AWS Toolkit-Version 3.69.0 oder höher erforderlich. Wenn Sie eine frühere Version des AWS Toolkits installiert haben, wird möglicherweise eine Meldung angezeigt. `Cannot open the handler` Um dieses Problem zu beheben, aktualisieren Sie Ihr AWS Toolkit auf die neueste Version.

Einzelheiten zur Verwendung AWS in Ihrer IDE finden Sie im AWS Toolkit for Visual Studio Code Benutzerhandbuch.

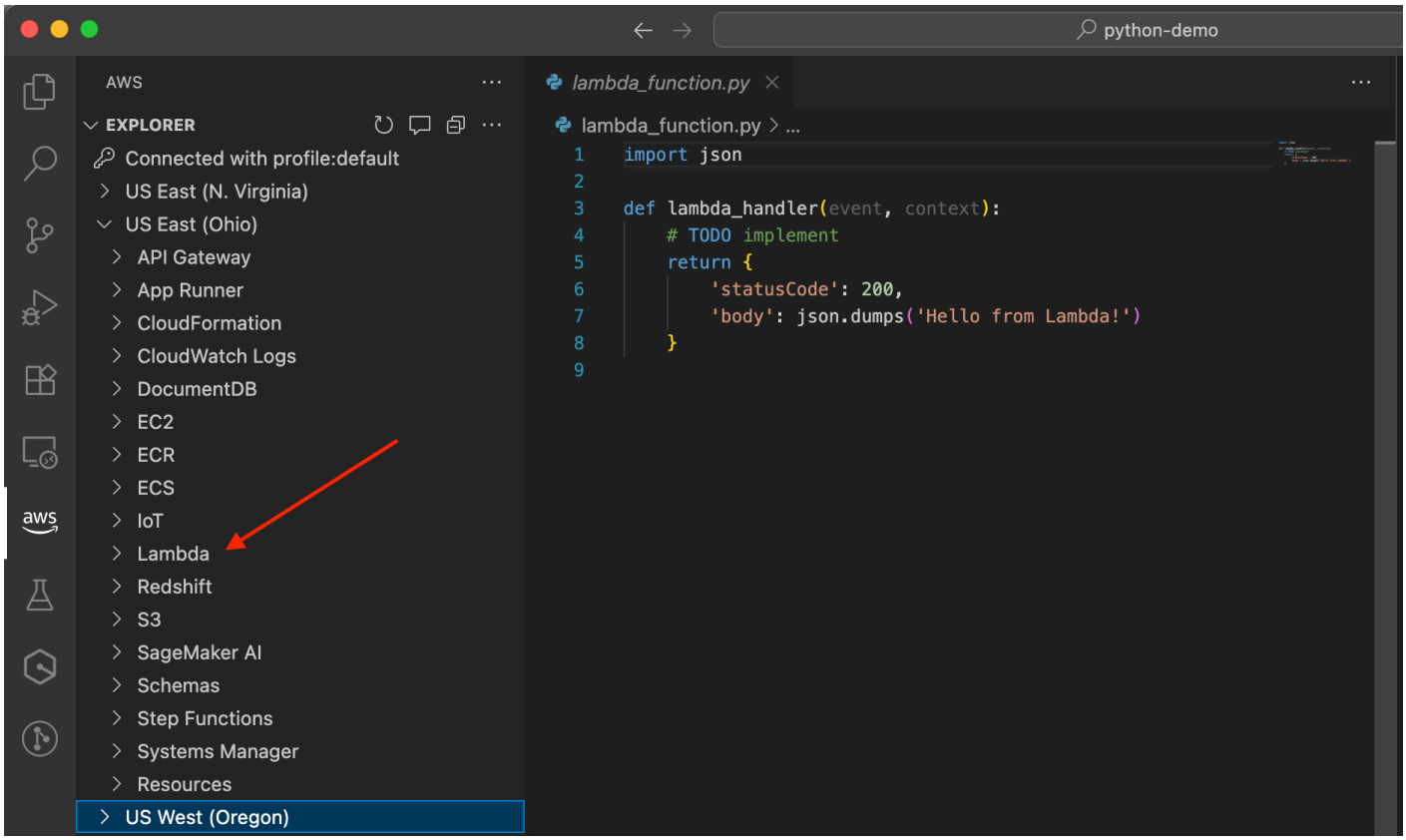
Lokales Arbeiten mit Funktionen

Nachdem Sie Ihre Funktion in Ihrer bevorzugten IDE geöffnet haben, folgen Sie diesen Schritten, um auf Ihre Funktionen zuzugreifen und sie zu verwalten. Für das folgende Beispiel haben wir Visual Studio Code verwendet.

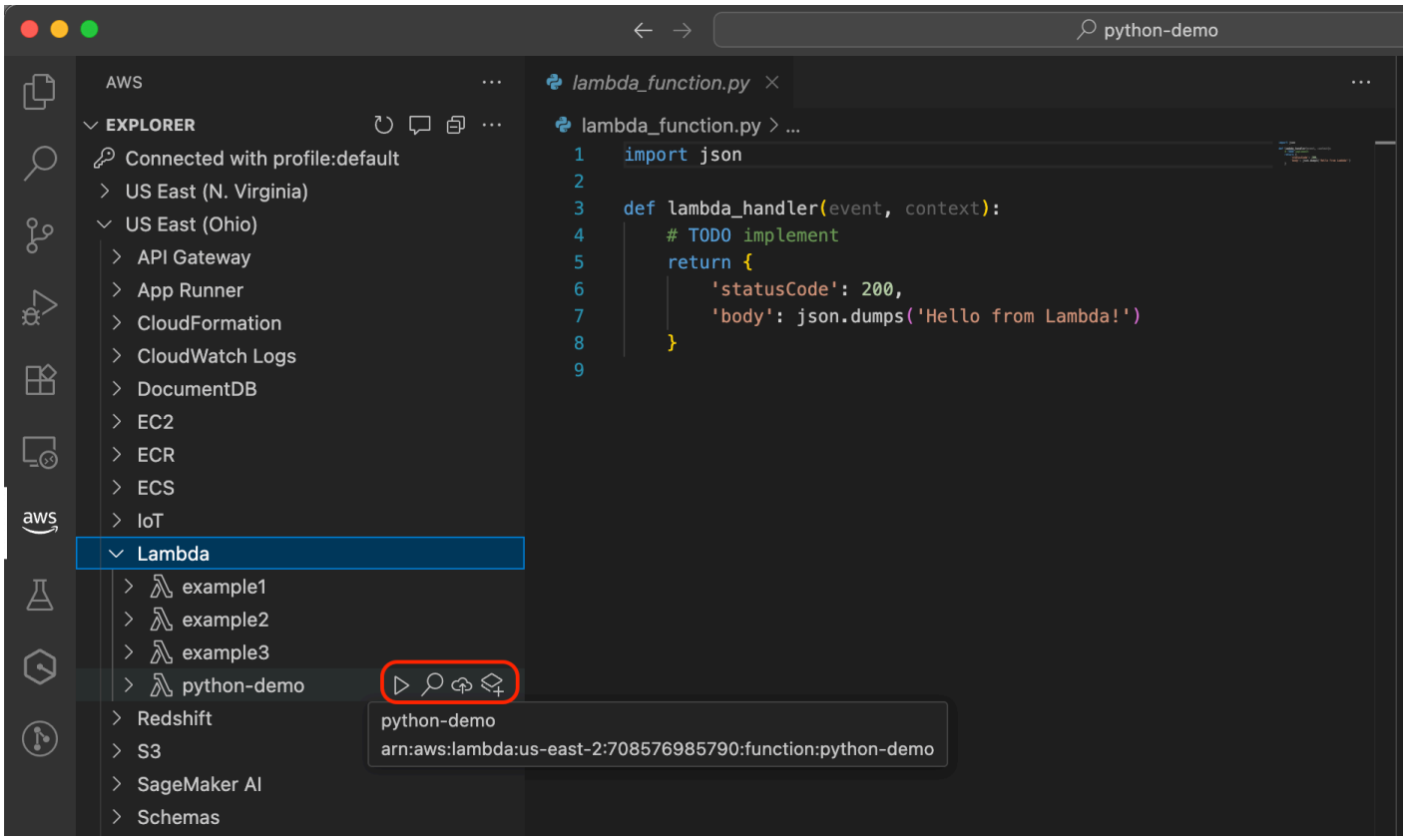
1. Wählen Sie das AWS Symbol in der Seitenleiste, um den AWS Explorer zu öffnen:



2. Wählen Sie im AWS Explorer die Region mit Ihrer Lambda-Funktion aus:



3. Erweitern Sie unter der ausgewählten Region den Abschnitt „Lambda“, um Ihre Funktionen anzuzeigen und zu verwalten:



Wenn Ihre Funktion in Ihrer IDE geöffnet ist, können Sie:

- Bearbeiten Sie den Funktionscode mit vollständiger Sprachunterstützung und Codevervollständigung.
- Verwenden Sie die LocalStack Integration, um Lambda-Funktionen zu testen, die AWS-Services während der Ausführung API-Aufrufe an andere richten, z. B. das Lesen aus DynamoDB-Tabellen oder das Schreiben in Amazon S3 S3-Buckets. LocalStack ist ein Cloud-Service-Emulator, der eine vollständige lokale Entwicklungsumgebung zum Testen von Serviceintegrationen bereitstellt. Sie können auch AWS SAM CLI verwenden, um Ihre Funktion in einem lokalen Container zu testen. Wenn Ihre Funktion API-Aufrufe an andere sendet AWS-Services, erreichen diese Aufrufe reale AWS Ressourcen, nicht emulierte.
- Debuggen Sie Ihre Funktion mit Haltepunkten und Variablenprüfung. Weitere Informationen finden Sie im AWS Toolkit for Visual Studio Code -Benutzerhandbuch unter Ausführen und Debuggen von Lambda-Funktionen direkt aus dem Code.
- Stellen Sie Ihre aktualisierte Funktion wieder AWS mithilfe des Cloud-Symbols bereit.

- Installieren und verwalten Sie Abhängigkeiten für Ihre Funktion.

Konvertiere deine Funktion in eine AWS SAM Vorlage erstellen und IaC-Tools verwenden

In allen unterstützten IDEs (Visual Studio Code, Kiro und Cursor) können Sie Ihre Lambda-Funktion in eine AWS SAM Vorlage konvertieren, indem Sie neben Ihrer Lambda-Funktion auf das Symbol In AWS SAM Anwendung konvertieren klicken. Sie werden aufgefordert, einen AWS SAM Projektstandort auszuwählen. Nach der Auswahl wird Ihre Lambda-Funktion in eine `template.yaml` Datei konvertiert, die in Ihrem neuen AWS SAM Projekt gespeichert wird.

Wenn Ihre Funktion in eine AWS SAM Vorlage konvertiert wurde, können Sie:

- Versionierung Ihrer Infrastruktur steuern
- Bereitstellungen automatisieren
- Funktionen aus der Ferne debuggen
- Fügen Sie Ihrer Anwendung zusätzliche AWS Ressourcen hinzu
- Während Ihres gesamten Entwicklungszyklus für konsistente Umgebungen sorgen
- Verwenden Sie Infrastructure Composer, um Ihre AWS SAM Vorlage visuell zu bearbeiten

Weitere Informationen zur Verwendung von IaC-Tools finden Sie in den folgenden Handbüchern:

- Das AWS Serverless Application Model Entwicklerhandbuch
- Das AWS Cloud Development Kit (AWS CDK) Entwicklerhandbuch
- Das Infrastructure-Composer-Entwicklerhandbuch
- Das AWS CloudFormation Benutzerhandbuch

Diese Tools bieten zusätzliche Funktionen zum Definieren, Testen und Bereitstellen Ihrer Serverless-Anwendungen.

Nächste Schritte

Weitere Informationen zum Arbeiten mit Lambda-Funktionen in Visual Studio Code finden Sie in den folgenden Ressourcen:

- Arbeiten mit AWS Lambda-Funktionen im AWS Toolkit for Visual Studio Code User Guide

- Arbeiten mit serverlosen Anwendungen im AWS Toolkit for Visual Studio Code User Guide
- Infrastructure as Code im Lambda-Benutzerhandbuch

Verwenden von GitHub Aktionen zum Bereitstellen von Ereignisfunktionen

Sie können GitHub Aktionen verwenden, um Lambda-Ereignisfunktionen automatisch bereitzustellen, wenn Sie Code- oder Konfigurationsänderungen in Ihr Repository übertragen. Die Aktion Lambda-Funktion bereitstellen bietet eine deklarative, einfache YAML-Schnittstelle, die die Komplexität manueller Bereitstellungsschritte beseitigt.

Beispiel-ArbeitsFlows

Um die automatisierte Bereitstellung von Lambda-Funktionen zu konfigurieren, erstellen Sie eine Workflow-Datei im `.github/workflows/`-Verzeichnis Ihres Repositorys:

Example GitHub Aktionsablauf für die Lambda-Bereitstellung

```
name: Deploy AWS Lambda
on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write # Required for OIDC authentication
      contents: read  # Required to check out the repository
    steps:
      - uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionRole
          aws-region: us-east-1

      - name: Deploy Lambda Function
        uses: aws-actions/aws-lambda-deploy@v1
        with:
```

```
          function-name: my-lambda-function
          code-artifacts-dir: ./dist
```

Dieser Workflow wird ausgeführt, wenn Sie Änderungen an den `main`-Branch übertragen. Es checkt Ihr Repository aus, konfiguriert AWS Anmeldeinformationen mit OpenID Connect (OIDC) und stellt Ihre Funktion mithilfe des Codes im Verzeichnis bereit. `./dist`

Weitere Beispiele wie die Aktualisierung der Funktionskonfiguration, die Bereitstellung über S3-Buckets und die Testlaufvalidierung finden Sie in der README-Datei Deploy Lambda Function.

Weitere Ressourcen

- Aktion „Anmeldeinformationen konfigurieren“ AWS GitHub
- Konfiguration von OpenID Connect in AWS

Verwenden von Lambda mit Infrastructure as Code (IaC)

Lambda-Ereignisfunktionen werden selten isoliert ausgeführt. Stattdessen sind sie oft Teil einer Serverless-Anwendung, zu der auch andere Ressourcen wie Datenbanken, Warteschlangen und Speicher gehören. Mit Infrastructure as Code (IaC) können Sie Ihre Bereitstellungsprozesse automatisieren, um ganze serverlose Anwendungen mit vielen separaten Ressourcen schnell und wiederholbar bereitzustellen und zu aktualisieren. AWS Dieser Ansatz beschleunigt Ihren Entwicklungszyklus, erleichtert die Konfigurationsverwaltung und stellt sicher, dass Ihre Ressourcen jedes Mal auf die gleiche Weise bereitgestellt werden.

IaC-Tools für Lambda

CloudFormation

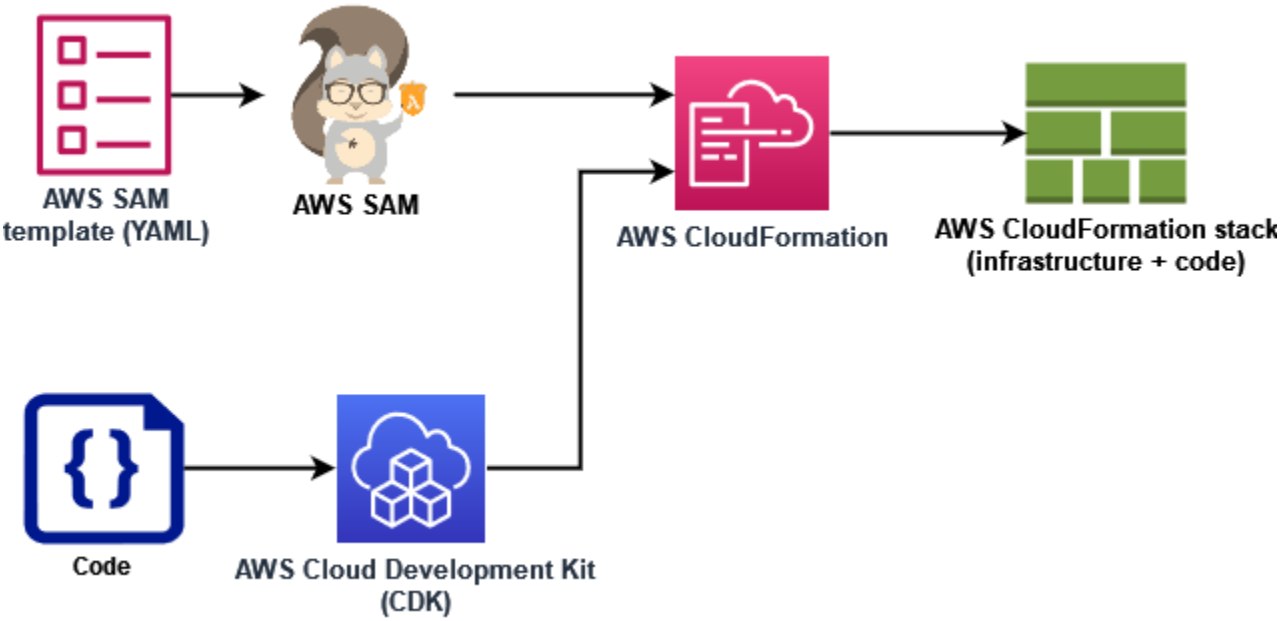
> CloudFormation ist der grundlegende IaC-Service von. AWS Sie können YAML- oder JSON-Vorlagen verwenden, um Ihre gesamte AWS Infrastruktur, einschließlich Lambda-Funktionen, zu modellieren und bereitzustellen. CloudFormation bewältigt die Komplexität beim Erstellen, Aktualisieren und Löschen Ihrer Ressourcen. AWS

AWS Serverless Application Model (AWS SAM)

> AWS SAM ist ein Open-Source-Framework, das darauf aufbaut. CloudFormation Es bietet eine vereinfachte Syntax für die Definition Serverless-Anwendungen. Verwenden Sie AWS SAM -Vorlagen, um Lambda-Funktionen, APIs, Datenbanken und Ereignisquellen schnell mit nur wenigen YAML-Zeilen bereitzustellen.

AWS Cloud Development Kit (AWS CDK)

Das CDK ist ein IaC-Ansatz, bei dem der Code an erster Stelle steht. Sie können Ihre Lambda-based Architektur mit TypeScript,, Python JavaScript, Java, C#/.Net oder Go definieren. Wählen Sie Ihre bevorzugte Sprache und verwenden Sie Programmierelemente wie Parameter, Konditionale, Schleifen, Komposition und Vererbung, um das gewünschte Ergebnis Ihrer Infrastruktur zu definieren. Das CDK generiert dann die zugrunde liegenden CloudFormation Vorlagen für die Bereitstellung. Ein Beispiel für die Verwendung von Lambda mit CDK finden Sie unter Bereitstellen von Ereignisfunktionen mit AWS CDK.



AWS bietet auch einen Dienst AWS Infrastructure Composer zur Entwicklung von IaC-Vorlagen mithilfe einer einfachen grafischen Oberfläche. Mit Infrastructure Composer entwerfen Sie eine Anwendungsarchitektur, indem Sie sie auf einer visuellen Leinwand ziehen, gruppieren und verbinden AWS-Services . Infrastructure Composer erstellt dann eine AWS SAM Vorlage oder eine CloudFormation Vorlage aus Ihrem Entwurf, die Sie zur Bereitstellung Ihrer Anwendung verwenden können.

Im folgenden Abschnitt the section called “Verwenden von AWS SAM und Infrastructure Composer” verwenden Sie Infrastructure Composer, um eine Vorlage für eine Serverless-Anwendung zu entwickeln, die auf einer vorhandenen Lambda-Funktion basiert.

Verwenden von Ereignisfunktionen in AWS SAM und Infrastructure Composer

In diesem Tutorial können Sie mit der Verwendung von IaC mit Lambda beginnen, indem Sie eine AWS SAM Vorlage aus einer vorhandenen Lambda-Funktion erstellen und dann eine serverlose Anwendung in Infrastructure Composer erstellen, indem Sie weitere Ressourcen hinzufügen. AWS

Während Sie dieses Tutorial durchführen, lernen Sie einige grundlegende Konzepte kennen, z. B. wie AWS Ressourcen in spezifiziert werden. AWS SAM Außerdem erfahren Sie, wie Sie mit Infrastructure Composer eine serverlose Anwendung erstellen, die Sie mit AWS SAM oder CloudFormation bereitstellen können.

Um dieses Tutorial abzuschließen, führen Sie die folgenden Schritte aus:

- Erstellen Sie eine Beispiel-Lambda-Funktion.
- Verwenden Sie die Lambda-Konsole, um die AWS SAM Vorlage für die Funktion anzuzeigen
- Exportieren Sie die Konfiguration Ihrer Funktion in AWS Infrastructure Composer und entwerfen Sie eine einfache serverlose Anwendung, die auf der Konfiguration Ihrer Funktion basiert
- Speichern Sie eine aktualisierte AWS SAM Vorlage, die Sie als Grundlage für die Bereitstellung Ihrer serverlosen Anwendung verwenden können

Voraussetzungen

In diesem Tutorial verwenden Sie die lokale Synchronisierungsfunktion von Infrastructure Composer, um Ihre Vorlagen- und Codedateien auf Ihrem lokalen Build-Computer zu speichern. Um diese Feature nutzen zu können, benötigen Sie einen Browser, der die File System Access-API unterstützt. Damit können Webanwendungen Dateien in Ihrem lokalen Dateisystem lesen, schreiben und speichern. Wir empfehlen, entweder Google Chrome oder Microsoft Edge zu verwenden. Weitere Informationen zur File System Access-API finden Sie unter Was ist die File System Access-API?.

Erstellen Sie eine Ereignisfunktion

In diesem ersten Schritt erstellen Sie eine Lambda-Funktion, mit der Sie den Rest des Tutorials abschließen können. Der Einfachheit halber verwenden Sie die Lambda-Konsole, um mithilfe der Python 3.11-Laufzeit eine grundlegende „Hello World“-Funktion zu erstellen.

Um mit der Konsole eine „Hello World“ -Ereignisfunktion zu erstellen

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie „Event-Funktion erstellen“.

3. Lassen Sie Ohne Vorgabe erstellen ausgewählt. Geben Sie im Feld Grundlegende Informationen für Funktionsname **`LambdaIaCDemo`** ein.
4. Wählen Sie für Laufzeit die Option Python 3.11 aus.
5. Wählen Sie „Event-Funktion erstellen“.

Sehen Sie sich das an AWS SAM Vorlage für Ihre Veranstaltung

Bevor Sie die Konfiguration Ihrer Funktion nach Infrastructure Composer exportieren, verwenden Sie die Lambda-Konsole, um die aktuelle Konfiguration Ihrer Funktion als AWS SAM -Vorlage anzuzeigen. Wenn Sie die Schritte in diesem Abschnitt befolgen, erfahren Sie mehr über den Aufbau einer AWS SAM Vorlage und darüber, wie Sie Ressourcen wie Lambda-Funktionen definieren, um mit der Spezifizierung einer serverlosen Anwendung zu beginnen.

Um das anzusehen AWS SAM Vorlage für Ihre Veranstaltung

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie gerade erstellt haben (`LambdaIaCDemo`).
3. Wählen Sie im Bereich Funktionsübersicht die Option Vorlage.

   Anstelle des Diagramms, das die Konfiguration Ihrer Funktion darstellt, wird eine AWS SAM Vorlage für Ihre Funktion angezeigt. Die Vorlage sollte jetzt wie folgt aussehen:

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
```

```
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

Nehmen wir uns einen Moment Zeit, um uns die YAML-Vorlage für Ihre Funktion anzusehen und einige wichtige Konzepte zu verstehen.

Die Vorlage beginnt mit der Deklaration `Transform: AWS::Serverless-2016-10-31`. Diese Deklaration ist erforderlich, da AWS SAM Vorlagen hinter den Kulissen bereitgestellt werden. CloudFormation Mithilfe der `Transform`-Anweisung wird die Vorlage als AWS SAM -Vorlagendatei identifiziert.

Nach der `Transform`-Deklaration folgt der `Resources`-Abschnitt. Hier werden die AWS Ressourcen definiert, die Sie mit Ihrer AWS SAM Vorlage bereitstellen möchten. AWS SAM Vorlagen können eine Kombination aus AWS SAM Ressourcen und CloudFormation Ressourcen enthalten. Das liegt daran, dass AWS SAM Vorlagen während der Bereitstellung zu CloudFormation Vorlagen erweitert werden, sodass jeder AWS SAM Vorlage jede gültige CloudFormation Syntax hinzugefügt werden kann.

Im Moment ist im Abschnitt `Resources` der Vorlage nur eine Ressource definiert, Ihre Lambda-Funktion `LambdaIaCDemo`. Um einer AWS SAM Vorlage eine Lambda-Funktion hinzuzufügen, verwenden Sie den `AWS::Serverless::Function` Ressourcentyp. Die `Properties` einer Lambda-Funktionsressource definieren die Laufzeit, den Funktionshandler und andere Konfigurationsoptionen der Funktion. Der Pfad zum Quellcode Ihrer Funktion, der für die Bereitstellung der Funktion verwendet werden AWS SAM soll, ist ebenfalls hier definiert. Weitere Informationen zu Lambda-Funktionsressourcen finden Sie unter AWS::Serverless: :Function im AWS SAM Developer Guide. AWS SAM

Neben den Funktionseigenschaften und Konfigurationen gibt die Vorlage auch eine AWS Identity and Access Management (IAM-) Richtlinie für Ihre Funktion an. Diese Richtlinie gibt Ihrer Funktion die Erlaubnis, Protokolle in Amazon CloudWatch Logs zu schreiben. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, hängt Lambda diese Richtlinie automatisch an Ihre Funktion an. Weitere Informationen zur Angabe einer IAM-Richtlinie für eine Funktion in einer AWS SAM Vorlage finden Sie in der entsprechenden `policies` Eigenschaft auf der Seite AWS::Serverless: :Function im AWS SAM Developer Guide.

Weitere Informationen zur Struktur von AWS SAM Vorlagen finden Sie unter AWS SAM Aufbau von Vorlagen.

Verwenden Sie AWS Infrastructure Composer um eine serverlose Anwendung zu entwerfen

Um mit der Erstellung einer einfachen serverlosen Anwendung zu beginnen, die die AWS SAM Vorlage Ihrer Funktion als Ausgangspunkt verwendet, exportieren Sie Ihre Funktionskonfiguration in Infrastructure Composer und aktivieren den lokalen Synchronisierungsmodus von Infrastructure Composer. Die lokale Synchronisierung speichert den Code Ihrer Funktion und Ihre AWS SAM Vorlage automatisch auf Ihrem lokalen Build-Computer und sorgt dafür, dass Ihre gespeicherte Vorlage synchronisiert wird, wenn Sie weitere AWS Ressourcen in Infrastructure Composer hinzufügen.

So exportieren Sie Ihre Funktion nach Infrastructure Composer

1. Wählen Sie im Bereich Funktionsübersicht die Option Nach Application Composer exportieren aus.

   Um die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren, erstellt Lambda einen Amazon-S3-Bucket in Ihrem Konto, um diese Daten vorübergehend zu speichern.

2. Wählen Sie im Dialogfeld Projekt bestätigen und erstellen aus, um den Standardnamen für diesen Bucket zu akzeptieren und die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren.
3. (Optional) Um einen anderen Namen für den von Lambda erstellten Amazon-S3-Bucket auszuwählen, geben Sie einen neuen Namen ein und wählen Sie Projekt bestätigen und erstellen aus. Die Amazon-S3-Bucket-Namen müssen global eindeutig sein und den Regeln für die Benennung von Buckets entsprechen.

   Wenn Sie Projekt bestätigen und erstellen auswählen, wird die Infrastructure-Composer-Konsole geöffnet. Auf der Leinwand sehen Sie Ihre Lambda-Funktion.
4. Wählen Sie in der Menü-Dropdown-Liste die Option Lokale Synchronisierung aktivieren aus.
5. Wählen Sie in dem sich öffnenden Dialogfeld die Option Ordner auswählen aus und wählen Sie einen Ordner auf Ihrem lokalen Build-Computer aus.
6. Wählen Sie Aktivieren aus, um die lokale Synchronisierung zu aktivieren.

Zum Exportieren Ihrer Funktion nach Infrastructure Composer benötigen Sie die Berechtigung zur Verwendung von bestimmten API-Aktionen. Wenn Sie Ihre Funktion nicht exportieren können, beachten Sie the section called “Erforderliche Berechtigungen” und stellen Sie sicher, dass Sie über die erforderlichen Berechtigungen verfügen.

> **Note**
>
> Für den Bucket, den Lambda erstellt, wenn Sie eine Funktion nach Infrastructure Composer exportieren, gelten die Amazon-S3-Standardpreise. Die Objekte, die Lambda in den Bucket einfügt, werden nach 10 Tagen automatisch gelöscht, aber Lambda löscht den Bucket selbst nicht.
> Folgen Sie den Anweisungen unter Löschen eines Buckets AWS-Konto, nachdem Sie Ihre Funktion in Infrastructure Composer exportiert haben, um zu vermeiden, dass Ihrem Konto zusätzliche Gebühren hinzugefügt werden. Weitere Informationen zum Amazon-S3-Bucket-Namen, den Lamda erstellt, finden Sie unter the section called “Infrastructure Composer”.

So entwerfen Sie Ihre Serverless-Anwendung in Infrastructure Composer

Nach der Aktivierung der lokalen Synchronisierung werden die Änderungen, die Sie in Infrastructure Composer vornehmen, in der AWS SAM Vorlage wiedergegeben, die auf Ihrem lokalen Build-

Computer gespeichert ist. Sie können jetzt zusätzliche AWS Ressourcen per Drag-and-Drop auf die Infrastructure Composer-Arbeitsfläche ziehen, um Ihre Anwendung zu erstellen. In diesem Beispiel fügen Sie eine einfache Amazon-SQS-Warteschlange als Trigger für Ihre Lambda-Funktion und eine DynamoDB-Tabelle für die Funktion hinzu, in die Daten geschrieben werden sollen.

1. Fügen Sie Ihrer Lambda-Funktion einen Amazon-SQS-Trigger hinzu, indem Sie wie folgt vorgehen:

   a. Geben Sie im Suchfeld in der Ressourcenpalette **`SQS`** ein.

   b. Ziehen Sie die Ressource SQS-Warteschlange auf Ihre Zeichenfläche und positionieren Sie sie links neben Ihrer Lambda-Funktion.

   c. Wählen Sie Details und geben Sie **`LambdaIaCQueue`** als Logische ID ein.

   d. Wählen Sie Speichern.

   e. Verbinden Sie Ihre Amazon-SQS- und Lambda-Ressourcen, indem Sie auf der SQS-Warteschlangenkarte auf den Port Abonnement klicken und ihn auf den linken Port auf der Lambda-Funktionskarte ziehen. Das Erscheinen einer Linie zwischen den beiden Ressourcen weist auf eine erfolgreiche Verbindung hin. Infrastructure Composer zeigt außerdem unten auf der Zeichenfläche eine Meldung an, die darauf hinweist, dass die beiden Ressourcen erfolgreich verbunden wurden.

2. Fügen Sie eine Amazon-DynamoDB-Tabelle für Ihre Lambda-Funktion hinzu, in die Daten geschrieben werden sollen, indem Sie so vorgehen:

   a. Geben Sie im Suchfeld in der Ressourcenpalette **`DynamoDB`** ein.

   b. Ziehen Sie die Ressource DynamoDB-Tabelle auf Ihre Zeichenfläche und positionieren Sie sie rechts neben Ihrer Lambda-Funktion.

   c. Wählen Sie Details und geben Sie **`LambdaIaCTable`** als Logische ID ein.

   d. Wählen Sie Speichern.

   e. Verbinden Sie die DynamoDB-Tabelle mit Ihrer Lambda-Funktion, indem Sie auf den rechten Port der Lambda-Funktionskarte klicken und ihn auf den linken Port der DynamoDB-Karte ziehen.

Nachdem Sie diese zusätzlichen Ressourcen hinzugefügt haben, werfen wir einen Blick auf die aktualisierte AWS SAM Vorlage, die Infrastructure Composer erstellt hat.

Um Ihr aktualisiertes zu sehen AWS SAM Vorlage

- Wählen Sie auf der Infrastructure-Composer-Zeichenfläche Vorlage, um von der Zeichenflächenansicht zur Vorlagenansicht zu wechseln.

Ihre AWS SAM Vorlage sollte jetzt die folgenden zusätzlichen Ressourcen und Eigenschaften enthalten:

- Eine Amazon-SQS-Warteschlange mit der ID `LambdaIaCQueue`

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

  Wenn Sie eine Amazon-SQS-Warteschlange mit Infrastructure Composer hinzufügen, setzt Infrastructure Composer die Eigenschaft `MessageRetentionPeriod`. Sie können die `FifoQueue`-Eigenschaft auch festlegen, indem Sie auf der SQS-Warteschlangenkarte Details auswählen und die Fifo-Warteschlange aktivieren oder deaktivieren.

  Um weitere Eigenschaften für Ihre Warteschlange festzulegen, können Sie die Vorlage manuell bearbeiten, um sie hinzuzufügen. Weitere Informationen zur `AWS::SQS::Queue`-Ressource und ihre verfügbaren Eigenschaften finden Sie unter AWS::SQS::Queue im CloudFormation - Benutzerhandbuch.

- Eine `Events`-Eigenschaft in Ihrer Lambda-Funktionsdefinition, die die Amazon-SQS-Warteschlange als Trigger für die Funktion angibt

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

  Die `Events`-Eigenschaft besteht aus einem Ereignistyp und einer Reihe von Eigenschaften, die vom Typ abhängen. Weitere Informationen zu den verschiedenen Konfigurationen, die AWS-Services Sie zum Auslösen einer Lambda-Funktion konfigurieren können, und

zu den Eigenschaften, die Sie festlegen können, finden Sie EventSourceim AWS SAM Entwicklerhandbuch.

- Eine DynamoDB-Tabelle mit dem Bezeichner `LambdaIaCTable`

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

Wenn Sie mit Infrastructure Composer eine DynamoDB-Tabelle hinzufügen, können Sie die Schlüssel Ihrer Tabelle festlegen, indem Sie auf der DynamoDB-Tabellenkarte Details auswählen und die Schlüsselwerte bearbeiten. Infrastructure Composer legt auch Standardwerte für eine Reihe anderer Eigenschaften fest, darunter `BillingMode` und `StreamViewType`.

Weitere Informationen zu diesen und anderen Eigenschaften, die Sie zu Ihrer AWS SAM Vorlage hinzufügen können, finden Sie unter AWS: :DynamoDB: :Table im Benutzerhandbuch.CloudFormation

- Eine neue IAM-Richtlinie, die Ihrer Funktion die Berechtigung gewährt, CRUD-Operationen für die von Ihnen hinzugefügte DynamoDB-Tabelle auszuführen.

```
Policies:
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

Die vollständige endgültige AWS SAM Vorlage sollte wie folgt aussehen.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
```

```
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
            BatchSize: 1
      Environment:
        Variables:
```

```
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

Stellen Sie Ihre serverlose Anwendung bereit mit AWS SAM (Optional)

Wenn Sie mithilfe der Vorlage AWS SAM , die Sie gerade in Infrastructure Composer erstellt haben, eine serverlose Anwendung bereitstellen möchten, müssen Sie zuerst die installieren. AWS SAM CLI Folgen Sie dazu den Anweisungen unter Installation der AWS SAM CLI.

Bevor Sie Ihre Anwendung bereitstellen, müssen Sie auch den Funktionscode aktualisieren, den Infrastructure Composer zusammen mit Ihrer Vorlage gespeichert hat. Derzeit enthält die von Infrastructure Composer gespeicherte `lambda_function.py`-Datei nur den grundlegenden „Hello World“-Code, den Lambda bei der Erstellung der Funktion bereitgestellt hat.

Um Ihren Funktionscode zu aktualisieren, kopieren Sie den folgenden Code und fügen Sie ihn in die `lambda_function.py`-Datei ein, die Infrastructure Composer auf Ihrer lokalen Build-Maschine gespeichert hat. Sie haben das Verzeichnis, in dem Infrastructure Composer diese Datei speichern soll, beim Aktivieren des Modus „Lokale Synchronisation“ angegeben.

Dieser Code akzeptiert ein Schlüssel-Wert-Paar in einer Nachricht von der Amazon-SQS-Warteschlange, die Sie in Infrastructure Composer erstellt haben. Wenn sowohl der Schlüssel als auch der Wert Zeichenfolgen sind, verwendet der Code diese Werte, um ein Element in die in Ihrer Vorlage definierte DynamoDB-Tabelle zu schreiben.

Aktualisierter Python-Funktionscode

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')

def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

So stellen Sie Ihre Serverless-Anwendung bereit

Gehen Sie wie folgt vor AWS SAM CLI, um Ihre Anwendung mit dem bereitzustellen. Damit Ihre Funktion korrekt erstellt und bereitgestellt werden kann, muss die Python-Version 3.11 auf Ihrem Build-Computer und in Ihrem `PATH` installiert sein.

1. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Infrastructure Composer Ihre Dateien `template.yaml` und `lambda_function.py` gespeichert hat.

   ```
   sam build
   ```

   Dieser Befehl sammelt die Build-Artefakte für Ihre Anwendung und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen.

2. Führen Sie den folgenden Befehl aus, um Ihre Anwendung bereitzustellen und die in Ihrer AWS SAM Vorlage angegebenen Lambda-, Amazon SQS- und DynamoDB-Ressourcen zu erstellen.

   ```
   sam deploy --guided
   ```

   Wenn Sie die `--guided` Markierung verwenden, AWS SAM werden Ihnen Eingabeaufforderungen angezeigt, die Sie durch den Bereitstellungsprozess führen. Akzeptieren Sie für diese Bereitstellung die Standardoptionen, indem Sie die Eingabetaste drücken.

AWS SAM Erstellt während des Bereitstellungsprozesses die folgenden Ressourcen in Ihrem AWS-Konto:

- Ein CloudFormation Stack mit dem Namen `sam-app`
- Eine Lambda-Funktion mit dem Namensformat `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`*
- Eine Amazon-SQS-Warteschlange mit dem Namensformat `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`*
- Eine DynamoDB-Tabelle mit dem Namensformat `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`*

AWS SAM erstellt auch die erforderlichen IAM-Rollen und -Richtlinien, sodass Ihre Lambda-Funktion Nachrichten aus der Amazon SQS SQS-Warteschlange lesen und CRUD-Operationen in der DynamoDB-Tabelle ausführen kann.

Testen der bereitgestellten Anwendung (optional)

Um zu überprüfen, ob Ihre Serverless-Anwendung korrekt bereitgestellt wurde, senden Sie eine Nachricht an Ihre Amazon-SQS-Warteschlange, die ein Schlüssel-Wert-Paar enthält, und überprüfen Sie, ob Lambda mit diesen Werten ein Element in Ihre DynamoDB-Tabelle schreibt.

So testen Sie Ihre Serverless-Anwendung

1. Öffnen Sie die Seite Warteschlangen der Amazon-SQS-Konsole und wählen Sie die Warteschlange aus, die AWS SAM aus Ihrer Vorlage erstellt hat. Der Name weist das Format `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`* auf.
2. Wählen Sie Nachrichten senden und empfangen aus und fügen Sie den folgenden JSON in den Nachrichtentext im Abschnitt Nachricht senden ein.

   ```
   {
       "myKey": "myValue"
   }
   ```
3. Klicken Sie auf Send Message (Nachricht senden).

   Wenn Sie Ihre Nachricht an die Warteschlange senden, ruft Lambda Ihre Funktion über Ihre Zuordnung von Ereignisquellen auf, die in Ihrer AWS SAM -Vorlage definiert ist. Um zu bestätigen, dass Lambda Ihre Funktion wie erwartet aufgerufen hat, vergewissern Sie sich, dass Ihrer DynamoDB-Tabelle ein Element hinzugefügt wurde.
4. Öffnen Sie die Seite Tabellen der DynamoDB-Konsole und wählen Sie Ihre Tabelle aus. Der Name weist das Format `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`* auf.
5. Wählen Sie Tabellenelemente erkunden aus. Im Bereich Zurückgegebene Elemente sollten Sie ein Element mit der ID `myKey` und dem Wert `myValue` sehen.

Bereitstellen von Ereignisfunktionen mit AWS CDK

Das AWS Cloud Development Kit (AWS CDK) ist ein Infrastructure-as-Code-Framework (IaC), mit dem Sie die AWS Cloud-Infrastruktur mithilfe einer Programmiersprache Ihrer Wahl definieren können. Um Ihre eigene Cloud-Infrastruktur zu definieren, schreiben Sie zunächst eine App (in einer der unterstützten Sprachen des CDK), die einen oder mehrere Stacks enthält. Anschließend synthetisieren Sie es zu einer CloudFormation Vorlage und stellen Ihre Ressourcen auf Ihrem bereit. AWS-Konto Folgen Sie den Schritten in diesem Thema, um eine Lambda-Ereignisfunktion bereitzustellen, die ein Ereignis von einem Amazon API Gateway Gateway-Endpunkt zurückgibt.

Die AWS Construct-Bibliothek, die im CDK enthalten ist, stellt Module bereit, mit denen Sie die bereitgestellten Ressourcen modellieren können. AWS-Services Für die gängigen Dienste stellt die Bibliothek kuratierte Konstrukte bereit, die intelligente Standards und bewährte Methoden bereitstellen. Sie können das Modul aws_lambda verwenden, um Ihre Funktion und die unterstützenden Ressourcen mit nur wenigen Codezeilen zu definieren.

Voraussetzungen

Bevor Sie mit diesem Tutorial beginnen, installieren Sie das, AWS CDK indem Sie den folgenden Befehl ausführen.

```
npm install -g aws-cdk
```

Schritt 1: Richten Sie Ihre ein AWS CDK project

Erstellen Sie ein Verzeichnis für Ihre neue AWS CDK App und initialisieren Sie das Projekt.

JavaScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language javascript
```

TypeScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language typescript
```

Python

```
mkdir hello-lambda
cd hello-lambda
cdk init --language python
```

Nach dem Start des Projekts aktivieren Sie die virtuelle Umgebung des Projekts und installieren die Basisabhängigkeiten für AWS CDK.

```
source .venv/bin/activate
```

```
python -m pip install -r requirements.txt
```

Java

```
mkdir hello-lambda
cd hello-lambda
cdk init --language java
```

Importieren Sie dieses Maven-Projekt in Ihre integrierte Java-Entwicklungsumgebung (IDE). Verwenden Sie in Eclipse beispielsweise Datei, Import, Maven, Vorhandene Maven-Projekte.

C#

```
mkdir hello-lambda
cd hello-lambda
cdk init --language csharp
```

> **Note**
>
> Die AWS CDK Anwendungsvorlage verwendet den Namen des Projektverzeichnisses, um Namen für Quelldateien und Klassen zu generieren. In diesem Beispiel trägt das Verzeichnis den Namen `hello-lambda`. Wenn Sie einen anderen Projektverzeichnisnamen verwenden, stimmt Ihre App nicht mit diesen Anweisungen überein.

AWS CDK v2 enthält stabile Konstrukte für alle AWS-Services in einem einzigen Paket, das aufgerufen wird`aws-cdk-lib`. Dieses Paket wird als Abhängigkeit installiert, wenn Sie das Projekt initialisieren. Wenn Sie mit bestimmten Programmiersprachen arbeiten, wird das Paket installiert, wenn Sie das Projekt zum ersten Mal erstellen.

Schritt 2: Definieren Sie AWS CDK Stack

Ein CDK-Stack ist eine Sammlung von einem oder mehreren Konstrukten, die Ressourcen definieren AWS . Jeder CDK-Stack steht für einen CloudFormation Stack in Ihrer CDK-App.

Um Ihren CDK-Stack zu definieren, folgen Sie den Anweisungen für Ihre bevorzugte Programmiersprache. Dieser Stack definiert das Folgende:

- Der logische Name der Funktion: `MyFunction`

- Der Speicherort des Funktionscodes, der in der `code`-Eigenschaft angegeben ist. Weitere Informationen finden Sie unter Handler-Code in der AWS Cloud Development Kit (AWS CDK) -API-Referenz.
- Der logische Name der REST-API: `HelloApi`
- Der logische Name des API-Gateway-Endpunkts: `ApiGwEndpoint`

Beachten Sie, dass alle CDK-Stacks in diesem Tutorial die Node.js Laufzeit für die Lambda-Funktion verwenden. Sie können verschiedene Programmiersprachen für den CDK-Stack und die Lambda-Funktion verwenden, um die Stärken der einzelnen Sprachen zu nutzen. Sie können TypeScript beispielsweise den CDK-Stack verwenden, um die Vorteile der statischen Typisierung für Ihren Infrastrukturcode zu nutzen. Sie können die Lambda-Funktion verwenden JavaScript , um die Flexibilität und schnelle Entwicklung einer dynamisch typisierten Sprache zu nutzen.

JavaScript

Öffnen Sie die Datei `lib/hello-lambda-stack.js` und ersetzen Sie den Inhalt durch Folgendes.

```
const { Stack } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigw = require('aws-cdk-lib/aws-apigateway');

class HelloLambdaStack extends Stack {
  /**
   *
   * @param {Construct} scope
   * @param {string} id
   * @param {StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);
    const fn = new lambda.Function(this, 'MyFunction', {
      code: lambda.Code.fromAsset('lib/lambda-handler'),
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler'
    });

    const endpoint = new apigw.LambdaRestApi(this, 'MyEndpoint', {
      handler: fn,
      restApiName: "HelloApi"
```

```
    });

  }
}

module.exports = { HelloLambdaStack }
```

TypeScript

Öffnen Sie die Datei `lib/hello-lambda-stack.ts` und ersetzen Sie den Inhalt durch Folgendes.

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as apigw from "aws-cdk-lib/aws-apigateway";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as path from 'node:path';

export class HelloLambdaStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps){
    super(scope, id, props)
    const fn = new lambda.Function(this, 'MyFunction', {
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler',
      code: lambda.Code.fromAsset(path.join(__dirname, 'lambda-handler')),
    });

    const endpoint = new apigw.LambdaRestApi(this, `ApiGwEndpoint`, {
      handler: fn,
      restApiName: `HelloApi`,
    });

  }
}
```

Python

Öffnen Sie die Datei `/hello-lambda/hello_lambda/hello_lambda_stack.py` und ersetzen Sie den Inhalt durch Folgendes.

```
from aws_cdk import (
    Stack,
```

```
    aws_apigateway as apigw,
    aws_lambda as _lambda
)
from constructs import Construct

class HelloLambdaStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        fn = _lambda.Function(
            self,
            "MyFunction",
            runtime=_lambda.Runtime.NODEJS_LATEST,
            handler="index.handler",
            code=_lambda.Code.from_asset("lib/lambda-handler")
        )

        endpoint = apigw.LambdaRestApi(
            self,
            "ApiGwEndpoint",
            handler=fn,
            rest_api_name="HelloApi"
        )
```

Java

Öffnen Sie die Datei `/hello-lambda/src/main/java/com/myorg/HelloLambdaStack.java` und ersetzen Sie den Inhalt durch Folgendes.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.lambda.Function;

public class HelloLambdaStack extends Stack {
    public HelloLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }
```

```
    public HelloLambdaStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Function hello = Function.Builder.create(this, "MyFunction")

 .runtime(software.amazon.awscdk.services.lambda.Runtime.NODEJS_LATEST)

 .code(software.amazon.awscdk.services.lambda.Code.fromAsset("lib/lambda-handler"))
                            .handler("index.handler")
                            .build();

        LambdaRestApi api = LambdaRestApi.Builder.create(this, "ApiGwEndpoint")
                                .restApiName("HelloApi")
                                .handler(hello)
                                .build();

    }
}
```

C#

Öffnen Sie die Datei `src/HelloLambda/HelloLambdaStack.cs` und ersetzen Sie den Inhalt durch Folgendes.

```
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Constructs;

namespace HelloLambda
{
    public class HelloLambdaStack : Stack
    {
        internal HelloLambdaStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var fn = new Function(this, "MyFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_LATEST,
                Code = Code.FromAsset("lib/lambda-handler"),
                Handler = "index.handler"
            });
```

```
            var api = new LambdaRestApi(this, "ApiGwEndpoint", new
 LambdaRestApiProps
            {
                Handler = fn
            });
        }
    }
}
```

Schritt 3: Erstellen Sie den Event-Funktionscode

1. Erstellen Sie im Stammverzeichnis Ihres Projekts (`hello-lambda`) das `/lib/lambda-handler`-Verzeichnis für den Lambda-Funktionscode. Dieses Verzeichnis ist in der `code` Eigenschaft Ihres AWS CDK Stacks angegeben.
2. Erstellen Sie in dem Verzeichnis `/lib/lambda-handler` eine neue Datei namens `index.js`. Fügen Sie folgenden Code in die Datei ein. Die Funktion extrahiert bestimmte Eigenschaften aus der API-Anfrage und gibt sie als JSON-Antwort zurück.

```
exports.handler = async (event) => {
  // Extract specific properties from the event object
  const { resource, path, httpMethod, headers, queryStringParameters, body } =
 event;
  const response = {
    resource,
    path,
    httpMethod,
    headers,
    queryStringParameters,
    body,
  };
  return {
    body: JSON.stringify(response, null, 2),
    statusCode: 200,
  };
};
```

Schritt 4: Stellen Sie das bereit AWS CDK Stack

1. Führen Sie im Stammverzeichnis Ihres Projekts den Befehl cdk synth aus:

```
cdk synth
```

Dieser Befehl synthetisiert eine AWS CloudFormation Vorlage aus Ihrem CDK-Stack. Die Vorlage ist eine ca. 400 Zeilen lange YAML-Datei, ähnlich der folgenden.

> **Note**
>
> Wenn Sie die folgende Fehlermeldung erhalten, vergewissern Sie sich, dass Sie sich im Stammverzeichnis Ihres Projektverzeichnisses befinden.
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

Example CloudFormation Vorlage

```
Resources:
  MyFunctionServiceRole3C357FF2:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: HelloLambdaStack/MyFunction/ServiceRole/Resource
  MyFunction1BAA52E7:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}
```

```
        S3Key:
 ab1111111cd32708dc4b83e81a21c296d607ff2cdef00f1d7f48338782f92l3901.zip
      Handler: index.handler
      Role:
        Fn::GetAtt:
          - MyFunctionServiceRole3C357FF2
          - Arn
      Runtime: nodejs24.x
      ...
```

2. Führen Sie den Befehl cdk deploy aus:

```
cdk deploy
```

Warten Sie, bis Ihre Ressourcen erstellt wurden. Die endgültige Ausgabe enthält die URL für Ihren API-Gateway-Endpunkt. Beispiel:

```
Outputs:
HelloLambdaStack.ApiGwEndpoint77F417B1 = https://abcd1234.execute-api.us-
east-1.amazonaws.com/prod/
```

Schritt 5: Testen der Funktion

Um die Lambda-Funktion aufzurufen, kopieren Sie den API Gateway-Endpunkt und fügen Sie ihn in einen Webbrowser ein oder führen Sie einen `curl`-Befehl aus:

```
curl -s https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/
```

Die Antwort ist eine JSON-Darstellung ausgewählter Eigenschaften aus dem ursprünglichen Ereignisobjekt, das Informationen über die Anfrage an den API-Gateway-Endpunkt enthält. Beispiel:

```
{
  "resource": "/",
  "path": "/",
  "httpMethod": "GET",
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
```

```
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-ASN": "16509",
    "CloudFront-Viewer-Country": "US",
    "Host": "abcd1234.execute-api.us-east-1.amazonaws.com",
     ...
```

Schritt 6: Bereinigen Ihrer Ressourcen

Der API-Gateway-Endpunkt ist öffentlich zugänglich. Um unerwartete Gebühren zu vermeiden, führen Sie den Befehl cdk destroy aus, um den Stack und alle zugehörigen Ressourcen zu löschen.

```
cdk destroy
```

Nächste Schritte

Informationen zum Schreiben von AWS CDK Apps in der Sprache Ihrer Wahl finden Sie im Folgenden:

TypeScript

- Arbeiten mit AWS CDK dem TypeScript

JavaScript

- Mit der Tinte AWS CDK arbeiten JavaScript

Python

- Arbeiten mit dem AWS CDK in Python

Java

- Arbeiten mit dem AWS CDK in Java

C#

- Arbeiten mit dem AWS CDK in C#

Go

- Arbeiten mit dem AWS CDK in Go

Elektrowerkzeuge für AWS Lambda

Powertools for AWS Lambda (auch als Powertools for bezeichnet AWS) bietet Hilfsfunktionen, Dekoratoren und Middleware, die allgemeine Lambda-Aufgaben wie strukturiertes Logging, Tracing, Metrikerfassung und Eingabevalidierung erledigen. Verwenden Sie Powertools for AWS Lambda, um serverlose Best Practices zu implementieren und die Entwicklung mehrerer Lambda Event Functions zu beschleunigen. Dadurch werden allgemeine Entwicklungsaufgaben in Ihren Lambda-Funktionen vereinfacht.

Hauptvorteile von Powertools für AWS

Die Lambda-Entwicklung ist zwar ohne Powertools for möglich AWS, ihre Verwendung bietet jedoch mehrere Vorteile:

- Built-in Beobachtbarkeit: Strukturierte Protokollierung, Rückverfolgung und benutzerdefinierte Metriken
- Verwaltung von Secrets: Abrufen von Parametern, Umgang mit Secrets und Idempotenz
- Progressive Verbesserung: Auswahl der Dienstprogramme, die Ihren Anforderungen am besten entsprechen
- Beschleunigte Entwicklung: Analyse, Validierung und Batchverarbeitung von Ereignissen
- Bewährte Methoden: Implementierung serverloser Muster AWS Well-Architected

Integration von Powertools mit AWS

Powertools for AWS hilft Ihnen dabei, produktionsreife serverlose Anwendungen mit weniger benutzerdefiniertem Code zu erstellen. Powertools for ist in Python TypeScript/Node.js, .NET und Java verfügbar und AWS kann über Lambda Layers oder mithilfe des Sprachpaketmanagers eingebunden werden. Jede Sprachimplementierung bietet wichtige Features wie strukturierte Protokollierung, Nachverfolgung, Erfassung von Metriken und Ereignisbehandlung, wobei die für jede Programmiersprache typischen Idiome beibehalten werden. Diese Implementierungen werden durch spezielle Komponenten für die AWS Serviceintegration ergänzt, die den Abruf von Parametern, die Stapelverarbeitung und die API-Handhabung unterstützen, sowie durch bewährte Methoden wie die Weitergabe von Korrelations-IDS, Fehlerbehandlung und Idempotenzmuster. Zusammen ermöglichen diese Features Entwicklern die Erstellung robuster, wartungsfreundlicher Serverless-Anwendungen bei gleichzeitig reduziertem Aufwand für benutzerdefinierten Code.

- Powertools für AWS Lambda (Python)

- Elektrowerkzeuge für AWS Lambda () TypeScript
- Powertools für AWS Lambda (Java)
- Powertools für AWS Lambda (.NET)

Nächste Schritte

Weitere Informationen zur Arbeit mit Powertools für finden Sie in den AWS folgenden Ressourcen:

- Workshop „Elektrowerkzeuge für AWS Lambda“
- Serverlose Muster, für die Powertools verwendet werden AWS
- AWS gut durchdachtes serverloses Objektiv
- Serverlose APIs mit Powertools für Lambda erstellen AWS

Verwaltung von Lambda-Workflows und -Ereignissen

Wenn Sie serverlose Anwendungen mit Lambda erstellen, benötigen Sie häufig Möglichkeiten, die Funktionsausführung zu orchestrieren und Ereignisse zu behandeln. AWS bietet mehrere Ansätze für die Koordination von Lambda-Ereignisfunktionen:

- Langlebige Lambda-Funktionen für die Code-First-Workflow-Orchestrierung innerhalb von Lambda
- AWS Step Functions für visuelle Workflow-Orchestrierung über mehrere Dienste hinweg
- Amazon EventBridge Scheduler und Amazon EventBridge für ereignisgesteuerte Architekturen und Planung

Sie können diese Ansätze auch zusammen integrieren. Beispielsweise können Sie den EventBridge Scheduler verwenden, um dauerhafte Funktionen oder Step Functions Functions-Workflows auszulösen, wenn bestimmte Ereignisse eintreten, oder Workflows so konfigurieren, dass Ereignisse an definierten Ausführungspunkten im EventBridge Scheduler veröffentlicht werden. Die folgenden Themen in diesem Abschnitt enthalten weitere Informationen dazu, wie Sie diese Orchestrierungsoptionen verwenden können.

Code-first Orchestrierung mit dauerhaften Funktionen

Dauerhafte Lambda-Funktionen bieten einen Code-First-Ansatz für die Workflow-Orchestrierung, sodass Sie statusbehaftete Workflows mit langer Laufzeit direkt in Ihren Lambda-Funktionen erstellen

können. Im Gegensatz zu externen Orchestrierungsdiensten behalten langlebige Funktionen Ihre Workflow-Logik im Code, was es einfacher macht, sie zusammen mit Ihrer Geschäftslogik zu versionieren, zu testen und zu warten.

Dauerhafte Funktionen sind ideal, wenn Sie Folgendes benötigen:

- Verwenden Sie Standardprogrammiersprachen: Definieren Sie Workflows mit vertrauten Programmiersprachen wie JavaScript Python
- Long-running Prozesse: Führen Sie Workflows aus, die bis zu einem Jahr lang ausgeführt werden können, also weit über dem standardmäßigen 15-minütigen Aufruflimit
- Vereinfachte Entwicklung: Behalten Sie die Workflow-Logik und die Geschäftslogik in derselben Codebasis, um die Wartung und das Testen zu vereinfachen
- Cost-effective Warten: Unterbrechen Sie die Ausführung während Wartezuständen, ohne Rechenressourcen zu verbrauchen
- Built-in Statusverwaltung: Automatisches Checkpointing und State-Persistenz ohne externe Speicherkonfiguration

Hilfe bei der Auswahl zwischen dauerhaften Funktionen und Step Functions finden Sie unter Dauerhafte Funktionen oder Step Functions.

Weitere Informationen zu dauerhaften Funktionen finden Sie unter Dauerhafte Lambda-Funktionen.

Orchestrierung von Workflows mit Step Functions

AWS Step Functions ist ein Workflow-Orchestrierungsservice, mit dem Sie mehrere Lambda-Funktionen und andere AWS Dienste in strukturierten Workflows koordinieren können. Diese Workflows können den Status beibehalten, Fehler mit ausgeklügelten Wiederholungsmechanismen behandeln und Daten in großem Umfang verarbeiten.

Step Functions bietet zwei Arten von Workflows für unterschiedliche Orchestrierungsanforderungen:

Standard-Workflows

> Ideal für lang laufende, überprüfbare Workflows, die eine Semantik erfordern, die nur einmal ausgeführt wird. Standard-Workflows können bis zu einem Jahr lang ausgeführt werden, bieten einen detaillierten Ausführungsverlauf und unterstützen visuelles Debugging. Sie eignen sich für Prozesse wie Auftragsabwicklung, Datenverarbeitungspipelines oder mehrstufige Analytikaufgaben.

Express-Workflows

Entwickelt für Workloads mit hoher Ereignisrate und kurzer Dauer mit Semantik, die mindestens einmal ausgeführt werden muss. Express-Workflows können bis zu fünf Minuten lang ausgeführt werden und eignen sich ideal für die Verarbeitung großer Ereignismengen, Streamingdatentransformationen oder IoT-Datenerfassungsszenarien. Sie bieten einen höheren Durchsatz und potenziell geringere Kosten im Vergleich zu Standard-Workflows.

> **Note**
>
> Weitere Informationen zu den Workflow-Typen von Step Functions finden Sie unter Auswahl des Workflow-Typs in Step Functions.

Innerhalb dieser Workflows bietet Step Functions zwei Arten von Kartenstatus für die parallele Verarbeitung:

Inline Map

Verarbeitet Elemente aus einem JSON-Array innerhalb des Ausführungsverlaufs des übergeordneten Workflows. Inline Map unterstützt bis zu 40 gleichzeitige Iterationen und eignet sich für kleinere Datensätze oder wenn Sie die gesamte Verarbeitung innerhalb einer einzigen Ausführung durchführen müssen. Weitere Informationen finden Sie unter Verwenden des Kartenstatus im Inline-Modus.

Distributed Map

Ermöglicht die Verarbeitung großer paralleler Workloads durch Iteration über Datensätze, die 256 KiB überschreiten oder mehr als 40 gleichzeitige Iterationen erfordern. Mit Unterstützung für bis zu 10 000 parallele untergeordnete Workflow-Ausführungen eignet sich Distributed Map hervorragend für die Verarbeitung von semistrukturierten Daten, die in Amazon S3 gespeichert sind, wie z. B. JSON- oder CSV-Dateien, und ist somit ideal für die Stapelverarbeitung und ETL-Vorgänge. Weitere Informationen finden Sie unter Verwenden des Kartenstatus im Distributed-Modus.

Durch die Kombination dieser Workflow-Typen und Kartenstatus bietet Step Functions ein flexibles und leistungsstarkes Toolset für die Orchestrierung komplexer Serverless-Anwendungen, von kleinen Vorgängen bis hin zu großen Datenverarbeitungspipelines.

Informationen zu den ersten Schritten mit Lambda mit Step Functions finden Sie unter Orchestrieren von Lambda-Funktionen mit Step Functions.

Verwaltung von Ereignissen mit EventBridge und Scheduler EventBridge

Amazon EventBridge ist ein Event-Bus-Service, der Sie beim Aufbau ereignisgesteuerter Architekturen unterstützt. Es leitet Ereignisse zwischen AWS Diensten, integrierten Anwendungen und SaaS-Anwendungen (Software as a Service) weiter. EventBridge Scheduler ist ein serverloser Scheduler. Mit EventBridge Scheduler können Sie Aufgaben von einem zentralen Dienst aus erstellen, ausführen und verwalten, Lambda-Funktionen nach einem Zeitplan mithilfe von Cron- und Rate-Ausdrücken aufrufen oder einmalige Aufrufe konfigurieren.

Amazon EventBridge und EventBridge Scheduler helfen Ihnen beim Aufbau ereignisgesteuerter Architekturen mit Lambda. EventBridge leitet Ereignisse zwischen AWS Diensten, integrierten Anwendungen und SaaS-Anwendungen weiter, während EventBridge Scheduler spezielle Planungsfunktionen für den wiederkehrenden oder einmaligen Aufruf von Lambda-Funktionen bietet.

Diese Services bieten mehrere wichtige Funktionen für die Arbeit mit Lambda-Funktionen:

- Erstellen Sie Regeln, die Ereignisse abgleichen und an Lambda-Funktionen weiterleiten, indem Sie EventBridge
- Richten Sie mit Scheduler wiederkehrende Funktionsaufrufe mithilfe von Cron- und Rate-Ausdrücken ein EventBridge
- Konfigurieren Sie einmalige Funktionsaufrufe zu bestimmten Daten und Uhrzeiten.
- Definieren Sie flexible Zeitfenster und Wiederholungsrichtlinien für geplante Aufrufe.

Weitere Informationen finden Sie unter Rufen Sie eine Lambda-Ereignisfunktion nach einem Zeitplan auf.

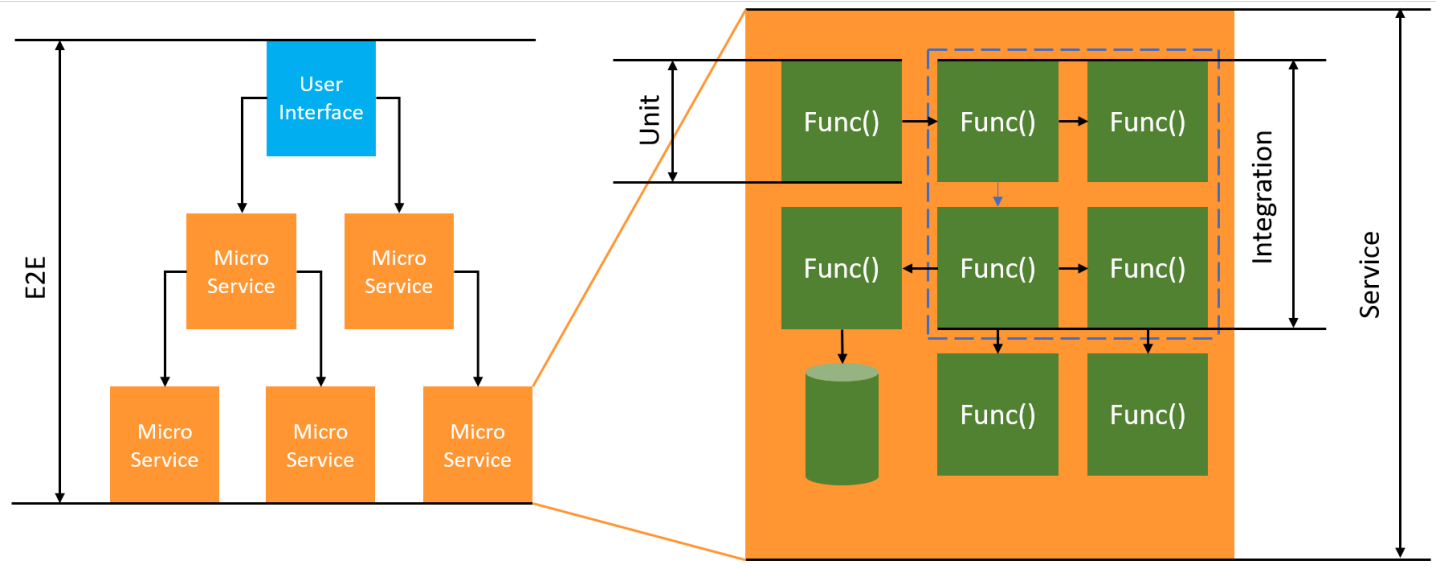
Wie man Serverless-Funktionen und Anwendungen testet

Beim Testen serverloser Funktionen werden herkömmliche Testtypen und -techniken verwendet, Sie müssen jedoch auch das Testen serverloser Anwendungen als Ganzes in Betracht ziehen. Cloud-based Tests bieten das genaueste Maß für die Qualität Ihrer Funktionen und serverlosen Anwendungen.

Eine Serverless-Anwendungsarchitektur umfasst verwaltete Services, die über API-Aufrufe wichtige Anwendungsfunktionen bereitstellen. Aus diesem Grund muss Ihr Entwicklungszyklus automatisierte Tests beinhalten, die bei der Interaktion Ihrer Funktionen und Services die Funktionalität überprüfen.

Wenn Sie keine cloud-basierten Tests erstellen, können aufgrund von Unterschieden zwischen Ihrer lokalen Umgebung und der bereitgestellten Umgebung Probleme auftreten. Ihr kontinuierlicher Integrationsprozess muss Tests anhand einer Reihe von Ressourcen durchführen, die in der Cloud bereitgestellt werden, bevor Ihr Code in die nächste Bereitstellungsumgebung wie QA, Staging oder Produktion übertragen wird.

Lesen Sie diesen kurzen Leitfaden weiter, um weitere Informationen zu Teststrategien für Serverless-Anwendungen zu erhalten, oder besuchen Sie das Serverless Test Samples Repository, um praktische Beispiele zu finden, die sich speziell auf die gewählte Sprache und Laufzeit beziehen.



Für serverlose Tests schreiben Sie immer noch Einheiten -, Integrations - und End-to-End-Tests.

- Einheitentests — Tests, die an einem isolierten Codeblock ausgeführt werden. Zum Beispiel die Überprüfung der Geschäftslogik zur Berechnung der Bereitstellungskosten für einen bestimmten Artikel und Bestimmungsort.
- Integrationstests — Tests, an denen zwei oder mehr Komponenten oder Dienste beteiligt sind, die interagieren, in der Regel in einer Cloud-Umgebung. Bei der Überprüfung einer Funktion werden beispielsweise Ereignisse aus einer Warteschlange verarbeitet.
- End-to-end Tests — Tests, die das Verhalten einer gesamten Anwendung überprüfen. Stellen Sie beispielsweise sicher, dass die Infrastruktur korrekt eingerichtet ist und die Ereignisse zwischen den Services wie erwartet ablaufen, um die Bestellungen der Kunden aufzuzeichnen.

Gezielte Geschäftsergebnisse

Das Testen serverloser Lösungen kann mehr Zeit in Anspruch nehmen, um es einzurichten. Sie müssen ereignisgesteuerte Interaktionen zwischen Diensten überprüfen. Beachten Sie beim Lesen dieses Leitfadens die folgenden praktischen geschäftlichen Gründe:

- Erhöhen Sie die Qualität Ihrer Anwendung
- Verkürzen Sie die Zeit für das Entwickeln von Features und das Beheben von Fehlern

Die Qualität einer Anwendung hängt vom Testen vieler Szenarien ab. Berücksichtigen Sie Ihre Geschäftsszenarien und automatisieren Sie Tests für die Ausführung mit Cloud-Diensten. Dies erhöht die Qualität Ihrer Anwendung.

Bugs und Konfigurationsprobleme kosten weniger, wenn sie zu Beginn des Entwicklungszyklus erkannt werden. Probleme, die bis zur Produktion unentdeckt bleiben, erfordern mehr Aufwand und mehr Mitarbeiter, um sie zu beheben.

Eine gute serverlose Teststrategie verbessert die Softwarequalität und beschleunigt Iterationen. Es überprüft, ob Ihre Lambda-Ereignisfunktionen und -anwendungen in der Cloud erwartungsgemäß funktionieren.

Was muss getestet werden?

Wir empfehlen eine Teststrategie, mit der das Verhalten der verwalteten Dienste, die Cloud-Konfiguration, die Sicherheitsrichtlinien und die Integration in Ihren Code getestet werden. Verhaltenstests, auch Blackbox-Tests genannt, stellen sicher, dass ein System erwartungsgemäß funktioniert, ohne dass die internen Abläufe bekannt sind.

- Führen Sie Einheitentests durch, um die Geschäftslogik innerhalb der Lambda-Funktionen zu überprüfen.
- Stellen Sie sicher, dass die integrierten Services tatsächlich aufgerufen werden und die Eingabeparameter korrekt sind.
- Stellen Sie sicher, dass ein Ereignis alle erwarteten Dienste in einem Workflow durchläuft.

In der traditionellen serverbasierten Architektur testen Teams oft nur den Code, der auf dem Anwendungsserver ausgeführt wird. Sie betrachten andere Komponenten, Dienste oder Abhängigkeiten als extern und außerhalb des Geltungsbereichs liegend.

Serverlose Anwendungen bestehen aus kleinen Arbeitseinheiten. Beispiele hierfür sind Lambda-Funktionen, die Produkte aus einer Datenbank abrufen, Elemente aus einer Warteschlange verarbeiten oder die Größe eines Bilds im Speicher ändern. Jede Komponente läuft in ihrer eigenen Umgebung. Teams verwalten viele dieser kleinen Einheiten in einer einzigen Anwendung.

Einige Funktionen können vollständig von Managed Services wie Amazon S3 abgewickelt oder ohne benutzerdefinierten Code erstellt werden. Sie müssen diese Managed Services nicht testen. Sie müssen jedoch testen, wie Ihr Code in sie integriert wird.

So wird Serverless getestet

Sie wissen wahrscheinlich, wie lokal bereitgestellte Anwendungen getestet werden. Sie schreiben Tests anhand von Code auf Ihrem Desktop oder in Containern. Sie können beispielsweise einen lokalen Webdienst aufrufen und dann die Antwort überprüfen.

Serverlose Lösungen verwenden Ihren Funktionscode und cloudbasierte verwaltete Dienste wie Warteschlangen, Datenbanken, Event-Busse und Messaging-Systeme. Diese Komponenten sind über eine ereignisgesteuerte Architektur miteinander verbunden, in der Nachrichten, sogenannte Ereignisse, von einer Ressource zur anderen fließen. Manche Interaktionen sind synchron, wie ein Webservice, der sofort Ergebnisse zurückgibt.

Andere sind asynchron, z. B. das Platzieren von Elementen in eine Warteschlange oder das Starten eines Workflow-Schritts. Ihre Teststrategie muss beide Typen abdecken und die Interaktionen zwischen Diensten testen. Bei asynchronen Interaktionen müssen Sie möglicherweise Nebenwirkungen in nachgeschalteten Komponenten erkennen, die nicht sofort sichtbar sind.

Sie können eine Cloud-Umgebung nicht vollständig lokal replizieren. Dazu gehören Warteschlangen, Datenbanktabellen, Event-Busse und Sicherheitsrichtlinien. Unterschiede zwischen lokalen und Cloud-Umgebungen führen zu Problemen. Diese Unterschiede verlängern die Zeit, um Fehler zu reproduzieren und zu beheben.

Bei serverlosen Anwendungen befinden sich die Komponenten vollständig in der Cloud. Tests mit Cloud-Code und -Diensten sind erforderlich, um Funktionen zu entwickeln und Fehler zu beheben.

Testmethoden

Ihre Teststrategie umfasst wahrscheinlich eine Mischung aus Techniken. Sie verwenden schnelle interaktive Tests, um Funktionen in der Konsole zu debuggen. Sie schreiben automatisierte Komponententests, um die Geschäftslogik zu überprüfen. Sie verifizieren Anrufe an externe

Dienste anhand von Mocks. Sie können auch Tests mit Emulatoren durchführen, die einen Dienst nachahmen.

- the section called “Testen in der Cloud”: Sie stellen Infrastruktur und Code bereit, um Tests mit tatsächlichen Diensten, Sicherheitsrichtlinien und Konfigurationen durchzuführen. Cloud-based Tests bieten das genaueste Maß für Ihre Codequalität.

  Das Debuggen einer Funktion in der Konsole ist eine schnelle Möglichkeit, in der Cloud zu testen. Sie können aus Beispieltestereignissen wählen oder ein benutzerdefiniertes Ereignis erstellen. Sie können Testereignisse auch über die Konsole mit Ihrem Team teilen.

  Um Tests im Entwicklungs- und Build-Lebenszyklus zu automatisieren, testen Sie außerhalb der Konsole. Automatisierungsstrategien finden Sie in den Abschnitten zu sprachspezifischen Tests in diesem Handbuch.
- the section called “Testen mit Mocks”: Mocks sind Objekte in Ihrem Code, die einen externen Dienst simulieren. Sie bieten vordefiniertes Verhalten zur Überprüfung von Serviceaufrufen und Parametern. Eine Fälschung ist eine Fälschung, bei der Abkürzungen verwendet werden, um das Testen zu vereinfachen oder zu beschleunigen. Beispielsweise könnte ein gefälschtes Datenzugriffsobjekt Daten aus einem In-Memory-Datenspeicher zurückgeben. Mocks können komplexe Abhängigkeiten vereinfachen, könnten aber dazu führen, dass mehr Mocks verschachtelte Abhängigkeiten ersetzen.
- the section called “Lokales Testen mit AWS SAM CLI”: Verwenden Sie AWS SAM CLI, um Lambda-Funktionen in Docker-Containern, die dieselbe Laufzeitumgebung wie Lambda verwenden, lokal aufzurufen. AWS Sie können die Funktionslogik und die Ereignisverarbeitung testen, ohne sie in der Cloud bereitstellen zu müssen.
- the section called “Testen mit Emulation”: Verwenden Sie die LocalStack Integration in VS Code, um mehrere AWS Dienste lokal zu emulieren, um Serviceintegrationen zu testen.

Testen in der Cloud

Das Testen in der Cloud ist für alle Testphasen wertvoll: Unit-Tests, Integrationstests und End-to-End-Tests. Tests, die mit cloudbasiertem Code und Diensten ausgeführt werden, liefern das genaueste Maß für Ihre Codequalität.

Eine einfache Möglichkeit, eine Lambda-Funktion in der Cloud auszuführen, ist ein Testereignis in der AWS-Managementkonsole. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe benötigt, kann das Ereignis ein leeres JSON-Dokument (`{}`) sein. Die

Konsole bietet Beispielereignisse für viele Serviceintegrationen. Sie können Ereignisse mit Ihrem Team teilen, um das Testen zu vereinfachen.

Erfahren Sie, wie Sie eine Beispielfunktion in der Konsole debuggen.

> **Note**
>
> Obwohl das Ausführen von Funktionen in der Konsole eine schnelle Möglichkeit zum Debuggen ist, ist die Automatisierung Ihrer Testzyklen unerlässlich, um die Anwendungsqualität und die Entwicklungsgeschwindigkeit zu erhöhen.

Beispiele für die Testautomatisierung sind im Serverless Test Samples Repository verfügbar. Mit der folgenden Befehlszeile wird ein Beispiel für einen Python-Integrationstest automatisch ausgeführt:

```
python -m pytest -s tests/integration -v
```

Obwohl der Test lokal ausgeführt wird, bezieht er sich auf cloudbasierte Ressourcen. Diese Ressourcen wurden mithilfe des AWS SAM Befehlszeilentools AWS Serverless Application Model und bereitgestellt. Der Testcode ruft zunächst die bereitgestellten Stack-Ausgaben ab, z. B. den API-Endpunkt, den Funktions-ARN und die Sicherheitsrolle.

Anschließend sendet er eine Anfrage an den API-Endpunkt. Die Antwort enthält eine Liste von Amazon S3 S3-Buckets. Dieser Test wird mit cloudbasierten Ressourcen ausgeführt, um zu überprüfen, ob sie bereitgestellt, gesichert und funktionsfähig sind.

```
========================= test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item


      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

      --> Stack outputs:
```

```
        HelloWorldApi
        = https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-east-2:123456789012:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::123456789012:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
      API Gateway response:
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

Für die Entwicklung cloudnativer Anwendungen bietet das Testen in der Cloud die folgenden Vorteile:

- Sie können jeden verfügbaren Service testen.
- Sie verwenden immer die neuesten Service-APIs und Rückgabewerte.
- Eine Cloud-Testumgebung ähnelt stark Ihrer Produktionsumgebung.
- Tests können Sicherheitsrichtlinien, Service Quotas, Konfigurationen und infrastrukturspezifische Parameter umfassen.
- Jeder Entwickler kann schnell eine oder mehrere Testumgebungen in der Cloud erstellen.
- Cloud-Tests erhöhen die Sicherheit, dass Ihr Code in der Produktion korrekt ausgeführt wird.

Das Testen in der Cloud hat einige Nachteile. Cloud-Bereitstellungen dauern in der Regel länger als lokale Desktop-Bereitstellungen.

Tools wie AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) Watch Mode und SST (3rd Party) reduzieren diese Latenz. Diese Tools überwachen Ihre Infrastruktur und Ihren Code und stellen dann automatisch Updates für Ihre Cloud-Umgebung bereit.

> **Note**
>
> Erfahren Sie im Serverless Developer Guide, wie Sie Infrastruktur als Code erstellen, um mehr über AWS Serverless Application Model CloudFormation, und AWS Cloud Development Kit (AWS CDK) zu erfahren.

Im Gegensatz zu lokalen Tests werden beim Cloud-Testen Ressourcen verwendet, für die möglicherweise Kosten anfallen. Isolierte Testumgebungen können für Ihre DevOps Teams zusätzliche Arbeit bedeuten, insbesondere in Organisationen mit strengen Kontokontrollen. Dennoch kann die Zeit der Entwickler für die Einrichtung einer komplexen lokalen Umgebung mehr kosten als die Verwendung von Cloud-Umgebungen, die mit Infrastructure-as-Code-Tools erstellt wurden.

Tests in der Cloud sind trotz dieser Überlegungen immer noch der beste Weg, um die Qualität Ihrer Serverless-Lösungen zu gewährleisten.

Testen mit Mocks

Das Testen mit Mocks ist eine Technik, bei der Sie Ersatzobjekte in Ihrem Code erstellen, um das Verhalten eines Cloud-Services zu simulieren.

Sie könnten beispielsweise einen Test schreiben, der ein Modell des Amazon S3 S3-Service verwendet. Der Mock gibt immer dann eine festgelegte Antwort zurück, wenn die CreateObjectMethode aufgerufen wird. Amazon S3 oder andere Service-Endpunkte werden nicht aufgerufen.

Schein-Frameworks generieren häufig Scheinobjekte für Sie. Einige Frameworks sind generisch. Andere zielen auf AWS SDKs ab, wie Moto, eine Python-Bibliothek zum Verspotten AWS von Diensten.

Scheinobjekte unterscheiden sich von Emulatoren. Entwickler erstellen Mocks als Teil des Testcodes. Emulatoren sind eigenständige Anwendungen, die dieselbe Funktionalität bieten wie die Systeme, die sie nachahmen.

Die Verwendung von Mocks bietet folgende Vorteile:

- Mocks können Dienste von Drittanbietern simulieren, die sich der Kontrolle Ihrer Anwendung entziehen, wie APIs und Software-as-a-Service (SaaS)-Anbieter, ohne dass ein direkter Zugriff auf diese Dienste erforderlich ist.

- Mocks sind nützlich für das Testen von Fehlerbedingungen, insbesondere wenn solche Bedingungen schwer zu simulieren sind, wie z. B. ein Service-Ausfall.
- Mocks können nach der Konfiguration schnelle lokale Tests ermöglichen.
- Mocks können Ersatzverhalten für praktisch jede Art von Objekt bereitstellen, so dass Mocking-Strategien eine breitere Palette von Diensten abdecken können als Emulatoren.
- Wenn neue Features oder Verhaltensweisen verfügbar werden, können Mock-Tests schneller reagieren. Durch die Verwendung eines generischen Mock-Frameworks können Sie neue Funktionen simulieren, sobald das aktualisierte AWS SDK verfügbar ist.

Mock-Tests haben die folgenden Nachteile:

- Mocks erfordern im Allgemeinen einen nicht unerheblichen Einrichtungs- und Konfigurationsaufwand, insbesondere wenn versucht wird, Rückgabewerte von verschiedenen Diensten zu ermitteln, um Antworten richtig zu simulieren.
- Mocks werden von Entwicklern geschrieben, konfiguriert und müssen von ihnen gewartet werden, was ihre Verantwortung erhöht.
- Möglicherweise benötigen Sie Zugriff auf die Cloud, um die APIs und Rückgabewerte von Diensten zu verstehen.
- Mocks können schwierig zu warten sein. Wenn sich die Signaturen der nachgebildeten Cloud-APIs ändern oder die Rückgabewertschemata weiterentwickelt werden, müssen Sie Ihre Mocks aktualisieren. Mocks erfordern auch Updates, wenn Sie Ihre Anwendungslogik erweitern, um Aufrufe an neue APIs zu tätigen.
- Tests, die Mocks verwenden, können in Desktop-Umgebungen erfolgreich sein, in der Cloud jedoch fehlschlagen. Die Ergebnisse stimmen möglicherweise nicht mit der aktuellen API überein. Die Servicekonfiguration und die Kontingente können nicht getestet werden.
- Schein-Frameworks können Richtlinien oder Kontingentbeschränkungen für AWS Identity and Access Management (IAM) nur begrenzt testen oder erkennen. Mocks können zwar besser simulieren, wenn die Autorisierung fehlschlägt oder ein Kontingent überschritten wird, aber Tests können nicht bestimmen, welches Ergebnis in einer Produktionsumgebung tatsächlich eintritt.

Lokales Testen mit AWS SAM CLI

Verwenden Sie AWS SAM CLI für das Testen Ihrer Funktionen in Docker-Containern mit derselben Laufzeitumgebung wie AWS Lambda. Sie können die Funktionslogik und die Ereignisverarbeitung lokal testen, ohne sie in der Cloud bereitstellen zu müssen. Wenn Ihre Funktion API-Aufrufe an andere sendet AWS-Services, erreichen diese Aufrufe reale AWS Ressourcen.

Tests mit lokalen Containern bieten unter anderem folgende Vorteile:

- Verwendet AWS Lambda Laufzeitumgebungen für genaue Tests.
- Ermöglicht schnelle lokale Entwicklungsiterationen ohne Cloud-Bereitstellung.
- Unterstützt das Debuggen mit vertrauten lokalen Entwicklungstools.

Das Testen mit lokalen Containern hat folgende Einschränkungen:

- AWS-Service Aufrufe Ihrer Funktion interagieren mit realen AWS Ressourcen, was Kosten verursachen und sich auf Produktionsdaten auswirken kann.
- Erfordert, dass Docker installiert ist und lokal ausgeführt wird.

Testen mit Emulation

Emulatoren sind lokal ausgeführte Anwendungen, die AWS Dienste nachahmen, indem sie ähnliche APIs und Rückgabewerte bereitstellen. LocalStack ist ein beliebtes Emulationstool, das eine vollständige lokale Entwicklungsumgebung zum Testen von Serviceintegrationen bietet.

LocalStack ist ein AWS Cloud-Emulator, mit dem Sie serverlose Anwendungen lokal testen können. Sie können Lambda-Funktionen testen, die in Dienste wie DynamoDB, Amazon S3 und Amazon SQS integriert sind, ohne eine Verbindung zu tatsächlichen Diensten herzustellen. AWS Sie können es LocalStack im AWS Toolkit for VS Code verwenden.

Tests mit Emulatoren bieten unter anderem folgende Vorteile:

- Emulatoren können dabei helfen, Iterationen und Tests bei der lokalen Entwicklung zu beschleunigen.
- Emulatoren bieten Entwicklern, die es gewohnt sind, Code in einer lokalen Umgebung zu entwickeln, eine vertraute Umgebung. Wenn Sie beispielsweise mit der Entwicklung einer n-Tier-Anwendung vertraut sind, verfügen Sie möglicherweise über eine Datenbank-Engine und einen Webserver, ähnlich denen, die in der Produktion laufen, auf Ihrem lokalen Computer, um schnelle, lokale, isolierte Testfunktionen bereitzustellen.
- Emulatoren erfordern keine Änderungen an der Cloud-Infrastruktur (z. B. Cloud-Konten für Entwickler), sodass sie einfach mit vorhandenen Testmustern implementiert werden können.
- Da Emulatoren keine tatsächlichen AWS Ressourcen verwenden, fallen keine unerwarteten Gebühren an, wenn Sie mehrere Dienste starten oder einige Ressourcen über einen längeren Zeitraum laufen lassen.

Das Testen mit Emulatoren hat folgende Nachteile:

- Emulatoren können schwierig einzurichten und zu replizieren sein, insbesondere wenn sie in Pipelines verwendet werden. CI/CD Dies kann den Workload von IT-Mitarbeitern oder Entwicklern erhöhen, die ihre eigene Software verwalten.
- Emulierte Features und APIs hinken in der Regel den Service-Updates hinterher. Dies kann zu Fehlern führen, da der getestete Code nicht mit der tatsächlichen API übereinstimmt, und die Einführung neuer Features behindern.
- Emulatoren benötigen Unterstützung, Updates, Bugfixes und Verbesserungen der Featureparität. Diese liegen in der Verantwortung des Emulatorautors, bei dem es sich um ein Drittunternehmen handeln kann.
- Tests, die auf Emulatoren basieren, können lokal erfolgreiche Ergebnisse liefern, schlagen jedoch in der Cloud aufgrund von Produktionssicherheitsrichtlinien, dienstübergreifenden Konfigurationen oder der Überschreitung von Lambda-Kontingenten fehl.

Bewährte Methoden

Die folgenden Abschnitte enthalten Empfehlungen für erfolgreiche Tests für Serverless-Anwendungen.

Praktische Beispiele für Tests und Testautomatisierung finden Sie im Serverless Test Samples Repository.

Priorisieren Sie Tests in der Cloud

Tests in der Cloud bieten die zuverlässigste, genaueste und vollständigste Testabdeckung. Bei der Durchführung von Tests im Cloud-Kontext werden nicht nur die Geschäftslogik, sondern auch Sicherheitsrichtlinien, Dienstkonfigurationen, Kontingente und die aktuellsten API-Signaturen und Rückgabewerte umfassend getestet.

Strukturieren Sie Ihren Code zum Testen

Vereinfachen Sie Ihre Tests und Lambda-Funktionen, indem Sie Lambda-specific Code von Ihrer Kerngeschäftslogik trennen.

Ihr Lambda-Funktions-Handler sollte ein schlanker Adapter sein, der Ereignisdaten aufnimmt und nur die Details an Ihre Geschäftslogikmethode(n) weiterleitet. Mit dieser Strategie können Sie umfassende Tests rund um Ihre Geschäftslogik durchführen, ohne sich Gedanken über Lambda-specific Details machen zu müssen. Ihre AWS Lambda-Funktionen sollten nicht die Einrichtung einer

komplexen Umgebung oder einer großen Anzahl von Abhängigkeiten erfordern, um die zu testende Komponente zu erstellen und zu initialisieren.

Im Allgemeinen sollten Sie einen Handler schreiben, der Daten aus den eingehenden Ereignis- und Kontext-Objekten extrahiert und validiert und diese Eingabe dann an Methoden sendet, die Ihre Geschäftslogik ausführen.

Entwicklungs-Feedback-Schleifen beschleunigen

Es gibt Tools und Techniken, um die Entwicklungs-Feedback-Schleifen zu beschleunigen. AWS SAM Accelerate und AWS CDK Watch Mode reduzieren beispielsweise beide die Zeit, die für die Aktualisierung von Cloud-Umgebungen benötigt wird.

In den Beispielen im GitHub Serverless Test Samples-Repository werden einige dieser Techniken untersucht.

Wir empfehlen außerdem, dass Sie Cloud-Ressourcen so früh wie möglich während der Entwicklung erstellen und testen — nicht erst, nachdem Sie sich bei der Quellverwaltung angemeldet haben. Diese Praxis ermöglicht ein schnelleres Erkunden und Experimentieren bei der Entwicklung von Lösungen. Darüber hinaus hilft Ihnen die Automatisierung der Bereitstellung von einem Entwicklungscomputer aus dabei, Probleme mit der Cloud-Konfiguration schneller zu erkennen und unnötigen Aufwand für Updates und Code-Review-Prozesse zu reduzieren.

Fokus auf Integrationstests

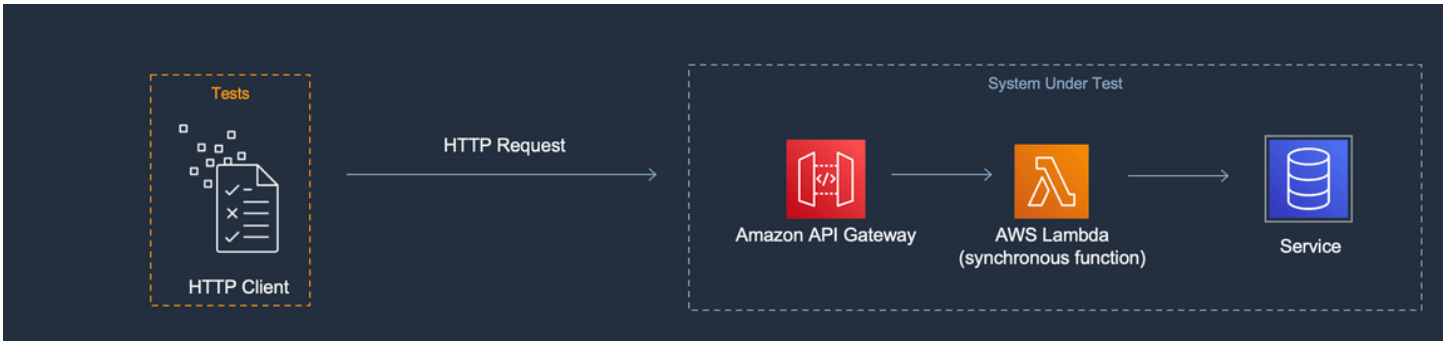
Beim Erstellen von Anwendungen mit Lambda hat es sich bewährt, Komponenten gemeinsam zu testen.

Tests, die mit zwei oder mehr Architekturkomponenten ausgeführt werden, werden als Integrationstests bezeichnet. Das Ziel von Integrationstests besteht nicht nur darin, zu verstehen, wie Ihr Code komponentenübergreifend ausgeführt wird, sondern auch, wie sich die Umgebung verhält, in der Ihr Code gehostet wird. End-to-end Tests sind spezielle Arten von Integrationstests, mit denen das Verhalten einer gesamten Anwendung überprüft wird.

Um Integrationstests zu erstellen, stellen Sie Ihre Anwendung in einer Cloud-Umgebung bereit. Dies kann von einer lokalen Umgebung aus oder über eine CI/CD Pipeline erfolgen. Schreiben Sie dann Tests, um das zu testende System (SUT) zu testen und das erwartete Verhalten zu validieren.

Das zu testende System könnte beispielsweise eine Anwendung sein, die API Gateway, Lambda und DynamoDB verwendet. Ein Test könnte einen synthetischen HTTP-Aufruf an einen API-Gateway-

Endpunkt ausführen und überprüfen, ob die Antwort die erwartete Nutzlast enthielt. Dieser Test bestätigt, dass der AWS Lambda-Code korrekt ist und dass jeder Dienst korrekt konfiguriert ist, um die Anfrage zu verarbeiten, einschließlich der IAM-Berechtigungen zwischen ihnen. Darüber hinaus könnten Sie den Test so gestalten, dass Datensätze unterschiedlicher Größe geschrieben werden, um zu überprüfen, ob Ihre Service-Kontingente, wie z. B. die maximale Datensatzgröße in DynamoDB, korrekt eingerichtet sind.



Erstellen Sie isolierte Testumgebungen

Tests in der Cloud erfordern in der Regel isolierte Entwicklerumgebungen, sodass sich Tests, Daten und Ereignisse nicht überschneiden.

Ein Ansatz besteht darin, jedem Entwickler ein eigenes Konto zur Verfügung zu stellen. AWS Dadurch werden Konflikte bei der Benennung von Ressourcen vermieden, die auftreten können, wenn mehrere Entwickler in einer gemeinsamen Codebasis arbeiten, versuchen, Ressourcen bereitzustellen oder eine API aufzurufen.

Automatisierte Testprozesse sollten für jeden Stapel eindeutig benannte Ressourcen erstellen. Sie können beispielsweise Skripts oder TOML-Konfigurationsdateien so einrichten, dass die AWS SAM CLI sam deploy- oder sam sync-Befehle automatisch einen Stack mit einem eindeutigen Präfix angeben.

In einigen Fällen teilen sich Entwickler ein AWS Konto. Dies kann daran liegen, dass Ihr Stack Ressourcen enthält, deren Betrieb oder Bereitstellung und Konfiguration teuer sind. Beispielsweise kann eine Datenbank gemeinsam genutzt werden, um die korrekte Einrichtung und das richtige Seeding der Daten zu vereinfachen

Wenn Entwickler ein Konto gemeinsam nutzen, müssen Sie Grenzen festlegen, um die Inhaberschaft zu ermitteln und Überschneidungen zu vermeiden. Eine Möglichkeit, dies zu tun, besteht darin, den Stack-Namen die Benutzer-IDs der Entwickler voranzustellen. Ein weiterer beliebter Ansatz ist das Einrichten von Stacks, die auf Code-Branches basieren. Aufgrund der Filialgrenzen sind die

Umgebungen isoliert, aber Entwickler können dennoch Ressourcen gemeinsam nutzen, z. B. eine relationale Datenbank. Dieser Ansatz ist eine bewährte Methode, wenn Entwickler an mehr als einer Verzweigung gleichzeitig arbeiten.

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Das Aufrechterhalten einer ordnungsgemäßen Isolierung ist von entscheidender Bedeutung. Dennoch sollten Sie darauf achten, dass Ihre QA-Umgebung der Produktionsumgebung so nahe wie möglich kommt. Aus diesem Grund fügen Teams den QA-Umgebungen Änderungssteuerungsvorgänge hinzu.

In Vorproduktions- und Produktionsumgebungen werden die Grenzen in der Regel auf Kontoebene gezogen, um Arbeitslasten von „Noisy-Neighbor“-Problemen zu isolieren und Sicherheitskontrollen mit geringsten Berechtigungen zum Schutz sensibler Daten zu implementieren. Für Workloads gibt es Kontingente. Ihre Tests dürfen weder die für die Produktion zugewiesenen Kontingente verbrauchen („Noisy Neighbor“) noch Zugriff auf Kundendaten haben. Lasttests sind eine weitere Aktivität, die Sie von Ihrem Produktions-Stack isolieren sollten.

In jedem Fall müssen die Umgebungen mit Warnungen und Kontrollen konfiguriert werden, um unnötige Ausgaben zu vermeiden. Sie können beispielsweise die Art, Stufe oder Größe der Ressourcen einschränken, die erstellt werden können, und E-Mail-Benachrichtigungen einrichten, wenn die geschätzten Kosten einen bestimmten Schwellenwert überschreiten.

Mocks für isolierte Geschäftslogik verwenden

Mock-Frameworks sind ein wertvolles Tool zum Schreiben schneller Einheitentests. Sie sind besonders nützlich, wenn Tests komplexe interne Geschäftslogiken wie mathematische oder finanzielle Berechnungen oder Simulationen abdecken. Suchen Sie nach Einheitentests, die eine große Anzahl von Testfällen oder Eingabevariationen enthalten, bei denen diese Eingaben das Muster oder den Inhalt von Aufrufen an andere Cloud-Services nicht ändern.

Code, der durch Komponententests mit Mocks abgedeckt wird, muss auch durch Tests in der Cloud abgedeckt werden. Dies wird empfohlen, da ein Entwickler-Laptop oder eine Build-Machine-Umgebung anders konfiguriert werden kann als eine Produktionsumgebung in der Cloud. Beispielsweise beanspruchen Ihre Lambda-Funktionen möglicherweise mehr Speicher oder Zeit als zugewiesen, wenn sie mit bestimmten Eingabeparametern ausgeführt werden. Möglicherweise enthält Ihr Code auch Umgebungsvariablen, die nicht auf die gleiche Weise (oder überhaupt nicht) konfiguriert sind, und die Unterschiede könnten dazu führen, dass sich der Code anders verhält oder fehlschlägt.

Bei Integrationstests ist der Nutzen von Mocks geringer, da der Aufwand für die Implementierung der erforderlichen Mocks mit der Anzahl der Verbindungspunkte zunimmt. End-to-end Beim Testen sollten keine Mocks verwendet werden, da sich diese Tests im Allgemeinen mit Zuständen und komplexer Logik befassen, die mit Schein-Frameworks nicht einfach simuliert werden können.

Vermeiden Sie schließlich die Verwendung von nachgebildeten Cloud-Services, um die ordnungsgemäße Implementierung von Service-Aufrufen zu überprüfen. Führen Sie stattdessen Cloud-Service-Aufrufe in der Cloud durch, um Verhalten, Konfiguration und funktionale Implementierung zu überprüfen.

Emulatoren sparsam verwenden

Emulatoren können für einige Anwendungsfälle praktisch sein, z. B. für ein Entwicklungsteam mit begrenztem, unzuverlässigem oder langsamem Internetzugang. In den meisten Fällen sollten Sie Emulatoren jedoch sparsam verwenden.

Indem Sie Emulatoren vermeiden, können Sie mit den neuesten Servicefunktionen und aktuellen APIs entwickeln und Innovationen entwickeln. Sie müssen nicht auf Herstellerversionen warten, um die gleiche Funktionalität zu erreichen. Sie reduzieren Ihre Vorab- und laufenden Kosten für den Kauf und die Konfiguration mehrerer Entwicklungssysteme und Build-Maschinen. Darüber hinaus vermeiden Sie das Problem, dass bei vielen Cloud-Diensten einfach keine Emulatoren verfügbar sind. Eine Teststrategie, die von Emulation abhängt, macht es unmöglich, diese Dienste zu nutzen (was zu potenziell teureren Behelfslösungen führt) oder Code und Konfigurationen zu erstellen, die nicht gut getestet wurden.

Wenn Sie die Emulation zum Testen verwenden, müssen Sie dennoch in der Cloud testen, um die Konfiguration zu überprüfen und Interaktionen mit Cloud-Diensten zu testen, die in einer emulierten Umgebung nur simuliert oder nachgeahmt werden können.

Herausforderungen beim Testen vor Ort

Wenn Sie Emulatoren und simulierte Aufrufe zum Testen auf Ihrem lokalen Desktop verwenden, kann es zu Testinkonsistenzen kommen, wenn Ihr Code in Ihrer Pipeline von Umgebung zu Umgebung fortschreitet. CI/CD Unit-Tests zur Validierung der Geschäftslogik Ihrer Anwendung auf Ihrem Desktop testen kritische Aspekte der Cloud-Dienste möglicherweise nicht genau.

Die folgenden Beispiele zeigen, worauf beim lokalen Testen mit Mocks und Emulatoren zu achten ist:

Beispiel: Die Event-Funktion erstellt einen S3-Bucket

Wenn die Logik einer Lambda-Funktion von der Erstellung eines S3-Buckets abhängt, sollte ein vollständiger Test bestätigen, dass Amazon S3 aufgerufen wurde und der Bucket erfolgreich erstellt wurde.

- In einem Test-Setup können Sie eine Erfolgsreaktion simulieren und möglicherweise einen Testfall hinzufügen, um eine Fehlerreaktion zu behandeln.
- In einem Emulationstestszenario kann die CreateBucketAPI aufgerufen werden, aber Sie müssen sich bewusst sein, dass die Identität, die den lokalen Aufruf durchführt, nicht vom Lambda-Dienst stammt. Die aufrufende Identität nimmt keine Sicherheitsrolle ein, wie dies in der Cloud der Fall wäre. Daher wird stattdessen eine Platzhalterauthentifizierung verwendet, möglicherweise mit einer toleranteren Rolle oder Benutzeridentität, die sich bei der Ausführung in der Cloud unterscheidet.

Die Mock- und Emulations-Setups testen, was die Lambda-Funktion tut, wenn sie Amazon S3 aufruft. Diese Tests verifizieren jedoch nicht, ob die Lambda-Funktion in der konfigurierten Form in der Lage ist, den Amazon S3 S3-Bucket erfolgreich zu erstellen. Sie müssen sicherstellen, dass der Rolle, die der Funktion zugewiesen ist, eine Sicherheitsrichtlinie angehängt ist, die es der Funktion ermöglicht, die `s3:CreateBucket`-Aktion auszuführen. Wenn nicht, schlägt die Funktion wahrscheinlich fehl, wenn sie in einer Cloud-Umgebung bereitgestellt wird.

Beispiel: Event Function verarbeitet Nachrichten aus einer Amazon SQS SQS-Warteschlange

Wenn eine Amazon-SQS-Warteschlange die Quelle einer Lambda-Funktion ist, muss durch einen vollständigen Test überprüft werden, ob die Lambda-Funktion erfolgreich aufgerufen wird, wenn eine Nachricht in eine Warteschlange gestellt wird.

Emulationstests und Scheintests sind im Allgemeinen so eingerichtet, dass der Lambda-Funktionscode direkt ausgeführt wird und die Amazon SQS SQS-Integration simuliert wird, indem eine JSON-Ereignisnutzlast (oder ein deserialisiertes Objekt) als Eingabe für den Funktionshandler übergeben wird.

Lokale Tests, die die Amazon SQS-Integration simulieren, testen, was die Lambda-Funktion tut, wenn sie von Amazon SQS mit einer bestimmten Nutzlast aufgerufen wird. Der Test bestätigt jedoch nicht, ob Amazon SQS die Lambda-Funktion erfolgreich aufruft, wenn sie in einer Cloud-Umgebung bereitgestellt wird.

Einige Beispiele für Konfigurationsprobleme, die bei Amazon SQS und Lambda auftreten können, sind die folgenden:

- Das Amazon-SQS-Sichtbarkeits-Timeout ist zu niedrig, was zu mehreren Aufrufen führt, obwohl nur einer vorgesehen war.
- Die Ausführungsrolle der Lambda-Funktion erlaubt nicht das Lesen von Nachrichten aus der Warteschlange (durch `sqs:ReceiveMessagesqs:DeleteMessage`, oder`sqs:GetQueueAttributes`).
- Das Beispielereignis, das an die Lambda-Funktion übergeben wird, überschreitet das Amazon-SQS-Nachrichtengrößenkontingent. Daher ist der Test ungültig, da Amazon SQS niemals in der Lage wäre, eine Nachricht dieser Größe zu senden.

Wie diese Beispiele zeigen, liefern Tests, die sich mit der Geschäftslogik, aber nicht mit den Konfigurationen zwischen Cloud-Services befassen, wahrscheinlich zu unzuverlässigen Ergebnissen.

Häufig gestellte Fragen

Ich habe eine Lambda-Funktion, die Berechnungen durchführt und ein Ergebnis zurückgibt, ohne andere Dienste aufzurufen. Muss ich es wirklich in der Cloud testen?

Ja. Lambda-Funktionen haben Konfigurationsparameter, die das Testergebnis verändern könnten. Der gesamte Lambda-Funktionscode ist von Timeout- und Speichereinstellungen abhängig, was dazu führen kann, dass die Funktion fehlschlägt, wenn diese Einstellungen nicht richtig gesetzt sind.

Lambda-Richtlinien ermöglichen auch die Standardausgabeprotokollierung an Amazon CloudWatch. Auch wenn Ihr Code nicht CloudWatch direkt aufruft, ist eine Genehmigung erforderlich, um die Protokollierung zu aktivieren. Diese erforderliche Erlaubnis kann nicht korrekt verspottet oder nachgeahmt werden.

Wie können Tests in der Cloud beim Einheitentest helfen? Wenn es sich in der Cloud befindet und eine Verbindung zu anderen Ressourcen herstellt, ist das nicht ein Integrationstest?

Wir definieren Komponententests als Tests, die isoliert an Architekturkomponenten ausgeführt werden. Dies verhindert jedoch nicht, dass Tests Komponenten einbeziehen, die möglicherweise andere Dienste aufrufen oder Netzwerkkommunikation verwenden.

Viele Serverless-Anwendungen verfügen über Architekturkomponenten, die isoliert getestet werden können, sogar in der Cloud. Ein Beispiel ist eine Lambda-Funktion, um Eingaben entgegenzunehmen, die Daten zu verarbeiten und eine Nachricht an eine Amazon-SQS-

Warteschlange zu senden. Ein Komponententest dieser Funktion würde wahrscheinlich testen, ob Eingabewerte dazu führen, dass bestimmte Werte in der Nachricht in der Warteschlange vorhanden sind.

Stellen Sie sich einen Test vor, der nach dem Muster „Arrange, Act, Assert“ geschrieben wurde:

- Arrange: Zuweisung von Ressourcen (eine Warteschlange zum Empfang von Nachrichten und die zu testende Funktion).
- Act: Aufrufen der zu testenden Funktion.
- Assert: Abrufen der von der Funktion gesendeten Nachricht und Validieren der Ausgabe.

Ein Mock-Testansatz würde das Mocking der Warteschlange mit einem prozessinternen Mock-Objekt und das Erstellen einer prozessinternen Instance der Klasse oder des Moduls, das den Lambda-Funktionscode enthält, beinhalten. Während der Assert-Phase würde die Nachricht in der Warteschlange aus dem simulierten Objekt abgerufen.

Bei einem cloud-basierten Ansatz würde der Test eine Amazon-SQS-Warteschlange für die Testzwecke erstellen und die Lambda-Funktion mit Umgebungsvariablen bereitstellen, die so konfiguriert sind, dass sie die isolierte Amazon-SQS-Warteschlange als Ausgabeziel verwenden. Nach dem Ausführen der Lambda-Funktion würde der Test die Nachricht aus der Amazon-SQS-Warteschlange abrufen.

Der cloudbasierte Test würde denselben Code ausführen, dasselbe Verhalten bestätigen und die funktionale Korrektheit der Anwendung überprüfen. Es hätte jedoch den zusätzlichen Vorteil, dass die Einstellungen der Lambda-Funktion validiert werden könnten: die IAM-Rolle, die IAM-Richtlinien sowie die Timeout- und Speichereinstellungen der Funktion.

Nächste Schritte und Ressourcen

Verwenden Sie die folgenden Ressourcen, um weitere Informationen und praktische Testbeispiele zu erhalten.

Beispielimplementierungen

Das Serverless Test Samples Repository on GitHub enthält konkrete Beispiele für Tests, die den in diesem Handbuch beschriebenen Mustern und bewährten Methoden folgen. Das Repository enthält Beispielcode und Anleitungen zu den Mock-, Emulations- und Cloud-Testprozessen, die in den vorherigen Abschnitten beschrieben wurden. Verwenden Sie dieses Repository, um sich über die neuesten Anleitungen zu serverlosen Tests von auf dem Laufenden zu halten. AWS

Weitere Informationen

Besuchen Sie Serverless Land, um auf die neuesten Blogs, Videos und Schulungen für AWS serverlose Technologien zuzugreifen.

Es wird auch empfohlen, die folgenden AWS Blogbeiträge zu lesen:

- Beschleunigen Sie die serverlose Entwicklung mit AWS SAM Accelerate (AWS Blogbeitrag)
- Erhöhung der Entwicklungsgeschwindigkeit mit CDK Watch (Blogbeitrag)AWS
- Serviceintegrationen mit AWS Step Functions Local verspotten (Blogbeitrag)AWS
- Erste Schritte beim Testen serverloser Anwendungen (Blogbeitrag)AWS

Tools

- AWS SAM — Testen und Debuggen serverloser Anwendungen
- AWS SAM — Integration mit automatisierten Tests
- Lambda — Testen von Lambda-Funktionen mit Hilfe der Lambda-Konsole

Verwaltung von Berechtigungen in AWS Lambda

Sie können AWS Identity and Access Management (IAM) verwenden, um Berechtigungen in AWS Lambda zu verwalten. Es gibt zwei Hauptkategorien von Berechtigungen, die Sie bei der Arbeit mit Lambda Event Functions berücksichtigen müssen:

- Berechtigungen, die Ihre Lambda-Funktionen benötigen, um API-Aktionen auszuführen und auf andere AWS Ressourcen zuzugreifen
- Berechtigungen, die andere AWS Benutzer und Entitäten für den Zugriff auf Ihre Lambda-Funktionen benötigen. Weitere Informationen finden Sie unter the section called “Zugriffsberechtigungen (Berechtigungen für andere Entitäten, auf Ihre Funktionen zuzugreifen)”.

Lambda-Funktionen müssen häufig auf andere AWS Ressourcen zugreifen und verschiedene API-Operationen mit diesen Ressourcen ausführen. Sie verfügen beispielsweise möglicherweise über eine Lambda-Funktion, die auf ein Ereignis reagiert, indem sie Einträge in einer Amazon-DynamoDB-Datenbank aktualisiert. In diesem Fall benötigt Ihre Funktion Berechtigungen für den Zugriff auf die Datenbank sowie Berechtigungen zum Ablegen oder Aktualisieren von Elementen in dieser Datenbank.

Sie definieren die Berechtigungen, die Ihre Lambda-Funktion benötigt, in einer speziellen IAM-Rolle, der sogenannten Ausführungsrolle. In dieser Rolle können Sie eine Richtlinie anhängen, die alle Berechtigungen definiert, die Ihre Funktion für den Zugriff auf andere AWS Ressourcen und das Lesen aus Ereignisquellen benötigt. Jede Lambda-Funktion muss eine Ausführungsrolle haben. Ihre Ausführungsrolle muss mindestens Zugriff auf Amazon haben, CloudWatch da Lambda-Funktionen standardmäßig in CloudWatch Logs protokolliert werden. Sie können die `AWSLambdaBasicExecutionRole`-verwaltete Richtlinie an Ihre Ausführungsrolle anhängen, um diese Anforderung zu erfüllen.

Um anderen AWS-Konten Organisationen und Diensten Zugriff auf Ihre Lambda-Ressourcen zu gewähren, haben Sie mehrere Möglichkeiten:

- Sie können identitätsbasierte Richtlinien verwenden, um anderen Benutzern Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Identity-based Richtlinien können direkt für Benutzer oder für Gruppen und Rollen gelten, die einem Benutzer zugeordnet sind.
- Sie können ressourcenbasierte Richtlinien verwenden, um anderen Konten und AWS-Services Berechtigungen für den Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Wenn ein Benutzer versucht, auf eine Lambda-Ressource zuzugreifen, berücksichtigt Lambda sowohl die identitätsbasierten Richtlinien des Benutzers als auch die ressourcenbasierte Richtlinie der Ressource. Wenn ein AWS Service wie Amazon Simple Storage Service (Amazon S3) Ihre Lambda-Funktion aufruft, berücksichtigt Lambda nur die ressourcenbasierte Richtlinie.
- Sie können ein attributbasiertes Zugriffskontrollmodell (ABAC) verwenden, um den Zugriff auf Ihre Lambda-Funktionen zu steuern. Mit ABAC können Sie Tags an eine Lambda-Funktion anhängen, sie in bestimmten API-Anforderungen übergeben oder sie an den IAM-Prinzipal anfügen, der die Anforderung stellt. Geben Sie dieselben Tags im Bedingungselement einer IAM-Richtlinie an, um den Funktionszugriff zu steuern.

In ist es eine bewährte Methode AWS, nur die Berechtigungen zu gewähren, die für die Ausführung einer Aufgabe erforderlich sind (Berechtigungen mit den geringsten Rechten). Um dies in Lambda zu implementieren, empfehlen wir, mit einer AWS -verwalteten Richtlinie zu beginnen. Sie können diese verwalteten Richtlinien unverändert oder als Ausgangspunkt zum Schreiben Ihrer eigenen restriktiveren Richtlinien verwenden.

Um Ihnen bei der Optimierung Ihrer Berechtigungen für den Zugriff mit den geringsten Rechten zu helfen, bietet Lambda einige zusätzliche Bedingungen, die Sie in Ihre Richtlinien aufnehmen können. Weitere Informationen finden Sie unter the section called “Ressourcen und Bedingungen”.

Weitere Informationen zu IAM finden Sie im IAM-Benutzerhandbuch.

Definieren von Lambda Event Function-Berechtigungen mit einer Ausführungsrolle

Die Ausführungsrolle einer Lambda-Ereignisfunktion ist eine AWS Identity and Access Management (IAM) -Rolle, die der Funktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. Sie könnten beispielsweise eine Ausführungsrolle erstellen, die berechtigt ist, Protokolle an Amazon zu senden CloudWatch und Trace-Daten hochzuladen AWS X-Ray. Diese Seite enthält Informationen zum Erstellen, Anzeigen und Verwalten der Ausführungsrolle einer Lambda-Funktion.

Lambda übernimmt automatisch Ihre Ausführungsrolle an, wenn Sie Ihre Funktion aufrufen. Sie sollten vermeiden, `sts:AssumeRole` manuell aufzurufen, um die Ausführungsrolle in Ihrem Funktionscode zu übernehmen. Wenn Ihr Anwendungsfall erfordert, dass die Rolle sich selbst annimmt, müssen Sie die Rolle selbst als vertrauenswürdigen Prinzipal in die Vertrauensrichtlinie Ihrer Rolle aufnehmen. Weitere Informationen zum Ändern einer Rollenvertrauensrichtlinie finden Sie unter Ändern einer Rollenvertrauensrichtlinie (Konsole) im IAM-Benutzerhandbuch.

Damit Lambda Ihre Ausführungsrolle ordnungsgemäß übernehmen kann, muss in der Vertrauensrichtlinie der Rolle der Lambda-Serviceprinzipal (`lambda.amazonaws.com`) als vertrauenswürdiger Service angegeben sein.

Themen

- Erstellen einer Ausführungsrolle in der IAM-Konsole
- Rollen erstellen und verwalten mit dem AWS CLI
- Gewähren Sie den Zugriff auf Ihre Lambda-Ausführungsrolle mit den geringsten Berechtigungen
- Berechtigungen in der Ausführungsrolle anzeigen und aktualisieren
- Arbeiten mit AWS verwalteten Richtlinien in der Ausführungsrolle
- Verwendung der Quellfunktion ARN zur Steuerung des Funktionszugriffsverhaltens

Erstellen einer Ausführungsrolle in der IAM-Konsole

Standardmäßig erstellt Lambda beim Erstellen einer Funktion in der Lambda-Konsole eine Ausführungsrolle mit minimalen Berechtigungen. Insbesondere umfasst diese Ausführungsrolle die `AWSLambdaBasicExecutionRole`verwaltete Richtlinie, die Ihrer Funktion grundlegende Berechtigungen zum Protokollieren von Ereignissen in Amazon CloudWatch Logs erteilt. Sie können im Abschnitt Berechtigungen die Option Standardrolle erstellen auswählen.

Sie können eine vorhandene Rolle auswählen, indem Sie im Abschnitt Berechtigungen die Option Andere Rolle verwenden auswählen. Wenn Ihre Lambda-Funktion zusätzliche Berechtigungen benötigt, um Aufgaben wie das Aktualisieren von Einträgen in einer Amazon DynamoDB DynamoDB-Datenbank als Reaktion auf Ereignisse auszuführen, können Sie eine benutzerdefinierte Ausführungsrolle mit den erforderlichen Berechtigungen erstellen. Wählen Sie dazu im Bereich Berechtigungen die Option Andere Rolle verwenden aus. Daraufhin wird eine Schublade geöffnet, in der Sie Ihre Berechtigungen anpassen können.

> **Note**
>
> Wenn der Rollenmanager in Ihrem Konto aktiviert ist, fügt die Lambda-Konsole eine Ausführungsrolle für Sie hinzu. Um Berechtigungen anzupassen, erstellen Sie Ihre Funktion und bearbeiten Sie dann die Berechtigungen der Funktion, um eine andere Rolle auszuwählen oder die zugeordnete Rolle anzupassen. Weitere Informationen finden Sie unter Automatisches Erstellen von Rollen mit dem Rollenmanager im IAM-Benutzerhandbuch.

So konfigurieren Sie eine Ausführungsrolle von der Konsole aus

1. Geben Sie im Abschnitt Rollendetails einen Rollennamen ein.
2. Wählen Sie im Abschnitt Richtlinie die Option Bestehende Richtlinie verwenden aus.
3. Wählen Sie die AWS verwalteten Richtlinien aus, die Sie Ihrer Rolle zuordnen möchten. Wenn Ihre Funktion beispielsweise auf DynamoDB zugreifen muss, wählen Sie die AWSLambdaDynamoDBExecutionRoleverwaltete Richtlinie aus.
4. Wählen Sie Rolle erstellen aus.

Wenn Sie eine Funktion in der Lambda-Konsole erstellen, können Sie der Funktion alternativ jede Ausführungsrolle zuordnen, die Sie zuvor erstellt haben. Wenn Sie einer vorhandenen Funktion eine neue Ausführungsrolle zuordnen möchten, folgen Sie den Schritten unter Aktualisieren der Ausführungsrolle einer Funktion.

Rollen erstellen und verwalten mit dem AWS CLI

Um eine Ausführungsrolle mit dem AWS Command Line Interface (AWS CLI) zu erstellen, verwenden Sie den create-role Befehl. Wenn Sie diesen Befehl verwenden, können Sie die Vertrauensrichtlinie angeben. Über die Vertrauensrichtlinie einer Rolle wird den angegebenen Prinzipalen die Berechtigung gegeben, die Rolle zu übernehmen. Im folgenden Beispiel erteilen Sie dem Lambda-

Serviceprinzipal die Berechtigung, Ihre Rolle zu übernehmen. Beachten Sie, dass die Anforderungen für das Maskieren von Anführungszeichen in der JSON-Zeichenfolge je nach Shell variieren können.

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document '{"Version": "2012-10-17",       "Statement":
 [{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action":
 "sts:AssumeRole"}]}'
```

Sie können die Vertrauensrichtlinie für die Rolle auch mithilfe einer separaten JSON-Datei definieren. Im folgenden Beispiel ist `trust-policy.json` eine Datei im aktuellen Verzeichnis.

Example trust-policy.json

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document file://trust-policy.json
```

Um der Rolle Berechtigungen hinzuzufügen, verwenden Sie den attach-policy-to-role-Befehl. Mit dem folgenden Befehl wird die verwaltete Richtlinie `AWSLambdaBasicExecutionRole` der Ausführungsrolle `lambda-ex` hinzugefügt.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

Nachdem Sie Ihre Ausführungsrolle erstellt haben, fügen Sie sie an Ihre Funktion an. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, können Sie eine beliebige Ausführungsrolle, die Sie zuvor erstellt haben, an die Funktion anhängen. Wenn Sie einer vorhandenen Funktion eine neue Ausführungsrolle zuordnen möchten, folgen Sie den Schritten unter the section called “Aktualisierung der Ausführungsrolle einer Funktion”.

Gewähren Sie den Zugriff auf Ihre Lambda-Ausführungsrolle mit den geringsten Berechtigungen

Wenn Sie während der Entwicklungsphase zum ersten Mal eine IAM-Rolle für Ihre Lambda-Funktion erstellen, können Sie manchmal Berechtigungen erteilen, die über das erforderliche Maß hinausgehen. Bevor Sie Ihre Funktion in der Produktionsumgebung veröffentlichen, sollten Sie als bewährte Methode die Richtlinie so anpassen, dass sie nur die erforderlichen Berechtigungen enthält. Weitere Informationen finden Sie im IAM-Benutzerhandbuch unter Anwenden von Berechtigungen mit den geringsten Rechten.

Verwenden Sie IAM Access Analyzer, um die erforderlichen Berechtigungen für die IAM-Ausführungsrollenrichtlinie zu identifizieren. IAM Access Analyzer überprüft Ihre AWS CloudTrail Protokolle über den von Ihnen angegebenen Zeitraum und generiert eine Richtlinienvorlage mit nur den Berechtigungen, die die Funktion in diesem Zeitraum verwendet hat. Sie können die Vorlage verwenden, um eine verwaltete Richtlinie mit definierten Berechtigungen zu erstellen und sie dann an die IAM-Rolle anzuhängen. Auf diese Weise gewähren Sie nur die Berechtigungen, die die Rolle für die Interaktion mit AWS Ressourcen für Ihren speziellen Anwendungsfall benötigt.

Weitere Informationen finden Sie im IAM-Benutzerhandbuch unter Generieren von Richtlinien auf der Grundlage von Zugriffsaktivitäten.

Berechtigungen in der Ausführungsrolle anzeigen und aktualisieren

In diesem Thema wird beschrieben, wie Sie die Ausführungsrolle Ihrer Funktion anzeigen und aktualisieren können.

Themen

- Anzeigen der Ausführungsrolle einer Funktion
- Aktualisierung der Ausführungsrolle einer Funktion

Anzeigen der Ausführungsrolle einer Funktion

Um die Ausführungsrolle einer Funktion anzuzeigen, verwenden Sie die Lambda-Konsole.

So zeigen Sie die Ausführungsrolle einer Funktion an (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und anschließend Permissions (Berechtigungen) aus.
4. Unter Ausführungsrolle können Sie die Rolle anzeigen, die derzeit als Ausführungsrolle der Funktion verwendet wird. Der Einfachheit halber können Sie im Abschnitt Ressourcenübersicht alle Ressourcen und Aktionen anzeigen, auf die die Funktion zugreifen kann. Sie können auch einen Dienst aus der Dropdown-Liste auswählen, um alle mit diesem Dienst verbundenen Berechtigungen anzuzeigen.

Aktualisierung der Ausführungsrolle einer Funktion

Sie können Berechtigungen jederzeit zur Ausführungsrolle einer Funktion hinzufügen bzw. daraus entfernen oder Ihre Funktion so konfigurieren, dass sie eine andere Rolle verwendet. Wenn Ihre Funktion Zugriff auf andere Dienste oder Ressourcen benötigt, müssen Sie der Ausführungsrolle die erforderlichen Berechtigungen hinzufügen.

Wenn Sie Ihrer Funktion Berechtigungen hinzufügen, führen Sie auch eine triviale Aktualisierung ihres Codes oder ihrer Konfiguration durch. Dies zwingt ausgeführte Instances Ihrer Funktion, die veraltete Anmeldeinformationen haben, anzuhalten und ersetzt zu werden.

Um die Ausführungsrolle einer Funktion zu aktualisieren, können Sie die Lambda-Konsole verwenden.

So aktualisieren Sie die Ausführungsrolle einer Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und anschließend Permissions (Berechtigungen) aus.
4. Wählen Sie unter Ausführungsrolle die Option Bearbeiten aus.
5. Wenn Sie Ihre Funktion so aktualisieren möchten, dass sie eine andere Rolle als Ausführungsrolle verwendet, wählen Sie die neue Rolle im Dropdownmenü unter Bestehende Rolle aus.

> **Note**
>
> Wenn Sie die Berechtigungen innerhalb einer vorhandenen Ausführungsrolle aktualisieren möchten, können Sie dies nur in der AWS Identity and Access Management (IAM-) Konsole tun.

Wenn Sie eine neue Rolle als Ausführungsrolle erstellen möchten, wählen Sie unter Ausführungsrolle die Option Neue Rolle aus AWS Richtlinienvorlagen erstellen aus. Geben Sie dann unter Rollenname einen Namen für Ihre neue Rolle ein und geben Sie unter Richtlinienvorlagen alle Richtlinien an, die Sie der neuen Rolle zuordnen möchten.

6. Wählen Sie Speichern.

Arbeiten mit AWS verwalteten Richtlinien in der Ausführungsrolle

Die folgenden AWS verwalteten Richtlinien bieten Berechtigungen, die für die Verwendung von Lambda-Funktionen erforderlich sind.

Änderung	Beschreibung	Datum
AWSLambdaMSKExecutionRolle — Lambda hat dieser Richtlinie die Berechtigung kafka: DescribeCluster V2 hinzugefügt.	`AWSLambdaMSKExecutionRole` gewährt Berechtigungen zum Lesen und Zugreifen auf Datensätze aus einem Amazon Managed Streaming for Apache Kafka (Amazon MSK) -Cluster, zum Verwalten elastischer Netzwerkschnittstellen (ENIs) und zum CloudWatch Schreiben in Protokolle.	17. Juni 2022
AWSLambdaBasicExecutionRole— Lambda hat damit begonnen, Änderungen an	`AWSLambdaBasicExecutionRole` erteilt Berechtigungen, um Protokolle auf CloudWatch hochzuladen.	14. Februar 2022

Änderung	Beschreibung	Datum
dieser Richtlinie nachzuver folgen.		
AWSLambdaDBExecuti onDynamo-Rolle — Lambda hat damit begonnen, Änderungen an dieser Richtlini e nachzuverfolgen.	`AWSLambdaDynamoDBE xecutionRole` gewährt Berechtigungen zum Lesen von Datensätzen aus einem Amazon DynamoDB DynamoDB-Stream und zum CloudWatch Schreiben in Logs.	14. Februar 2022
AWSLambdaKinesisEx ecutionRole— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuver folgen.	`AWSLambdaKinesisEx ecutionRole` gewährt Berechtigungen zum Lesen von Ereignissen aus einem Amazon Kinesis Kinesis-Datenstream und zum Schreiben in CloudWatch Protokolle.	14. Februar 2022
AWSLambdaMSKExecut ionRolle — Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuver folgen.	`AWSLambdaMSKExecut ionRole` gewährt Berechtig ungen zum Lesen und Zugreifen auf Datensätze aus einem Amazon Managed Streaming for Apache Kafka (Amazon MSK) -Cluster, zum Verwalten elastisch er Netzwerkschnittstellen (ENIs) und zum CloudWatch Schreiben in Protokolle.	14. Februar 2022

Änderung	Beschreibung	Datum
AWSLambdaSQSQueueExecutionRole— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AWSLambdaSQSQueueExecutionRole` gewährt Berechtigungen zum Lesen einer Nachricht aus einer Amazon Simple Queue Service (Amazon SQS) - Warteschlange und zum Schreiben in CloudWatch Protokolle.	14. Februar 2022
AWSLambdaVPCAccessExecutionRole— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AWSLambdaVPCAccessExecutionRole` gewährt Berechtigungen zum Verwalten ENIs innerhalb einer Amazon VPC und zum Schreiben in CloudWatch Logs.	14. Februar 2022
AWSXRayDaemonWriteAccess— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AWSXRayDaemonWriteAccess` erteilt Berechtigungen zum Hochladen von Nachverfolgungsdaten auf X-Ray.	14. Februar 2022
CloudWatchLambdaInsightsExecutionRolePolicy— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`CloudWatchLambdaInsightsExecutionRolePolicy` gewährt Berechtigungen zum Schreiben von Laufzeitmetriken in CloudWatch Lambda Insights.	14. Februar 2022

Änderung	Beschreibung	Datum
AmazonS3 ObjectLambdaExecutionRolePolicy — Lambda hat begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AmazonS3ObjectLambdaExecutionRolePolicy` gewährt Berechtigungen zur Interaktion mit dem Amazon Simple Storage Service (Amazon S3) -Objekt Lambda und zum Schreiben in CloudWatch Logs.	14. Februar 2022

Bei einigen Funktionen versucht die Lambda-Konsole, Ihrer Ausführungsrolle in einer vom Kunden verwalteten Richtlinie fehlende Berechtigungen hinzuzufügen. Diese Politiken können zahlreich werden. Fügen Sie der Ausführungsrolle die relevanten AWS -verwalteten Richtlinien hinzu, bevor Sie Funktionen aktivieren, um das Erstellen zusätzlicher Richtlinien zu vermeiden.

Wenn Sie eine Ereignisquellen-Zuweisung zum Aufrufen Ihrer Funktion verwenden, verwendet Lambda die Ausführungsrolle zum Lesen von Ereignisdaten. Zum Beispiel liest eine Ereignisquellen-Zuweisung für Kinesis-Ereignisse aus einem Datenstrom und sendet diese in Batches an Ihre Funktion.

Wenn ein Dienst eine Rolle in Ihrem Konto übernimmt, können Sie die globalen Bedingungskontextschlüssel `aws:SourceAccount` und `aws:SourceArn` in Ihrer Rollenvertrauensrichtlinie übernehmen, um den Zugriff auf die Rolle nur auf Anforderungen zu beschränken, die von erwarteten Ressourcen generiert werden. Weitere Informationen finden Sie unter Vermeidung des Problems des verwirrten Stellvertreters im dienstübergreifenden Szenario für AWS -Security-Token-Service.

Zusätzlich zu den AWS verwalteten Richtlinien bietet die Lambda-Konsole Vorlagen für die Erstellung einer benutzerdefinierten Richtlinie mit Berechtigungen für zusätzliche Anwendungsfälle. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, haben Sie die Wahl, eine neue Ausführungsrolle mit Berechtigungen aus einer oder mehreren Vorlagen zu erstellen. Diese Vorlagen werden auch automatisch angewendet, wenn Sie eine Funktion von einer Vorlage erstellen oder wenn Sie Optionen konfigurieren, die Zugriff auf andere Services erfordern. Beispielvorlagen sind im GitHubRepository dieses Handbuchs verfügbar.

Verwendung der Quellfunktion ARN zur Steuerung des Funktionszugriffsverhaltens

Es ist üblich, dass Ihr Lambda Event Function-Code API-Anfragen an andere AWS-Services sendet. Um diese Anforderungen zu stellen, generiert Lambda einen kurzlebigen Satz von Anmeldeinformationen, indem es die Ausführungsrolle Ihrer Funktion übernimmt. Diese Anmeldeinformationen sind während des Aufrufs Ihrer Funktion als Umgebungsvariablen verfügbar. Wenn Sie mit AWS -SDKs arbeiten, müssen Sie die Anmeldeinformationen für das SDK nicht direkt im Code angeben. Standardmäßig überprüft die Kette der Anbieter von Anmeldeinformationen nacheinander jeden Ort, an dem Sie Anmeldeinformationen festlegen können, und wählt die erste verfügbare aus – in der Regel die Umgebungsvariablen (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` und `AWS_SESSION_TOKEN`).

Lambda fügt die Quellfunktion ARN in den Anmeldeinformationskontext ein, wenn es sich bei der Anfrage um eine AWS API-Anforderung handelt, die aus Ihrer Ausführungsumgebung stammt. Lambda fügt auch den Quellfunktions-ARN für die folgenden AWS -API-Anfragen ein, die Lambda in Ihrem Namen außerhalb Ihrer Ausführungsumgebung durchführt:

Service	Action	Grund
CloudWatch Logs	`CreateLogGroup` , `CreateLogStream` , `PutLogEvents`	Um Protokolle in einer CloudWatch Logs-Prot okollgruppe zu speichern
X-Ray	`PutTraceSegments`	Um Ablaufverfolgungsdaten zu senden X-Ray
Amazon EFS	`ClientMount`	So stellen Sie eine Verbindun g zwischen Ihrer Funktion und einem Amazon Elastic File System (Amazon EFS) Dateisystem her

Andere AWS API-Aufrufe, die Lambda außerhalb Ihrer Ausführungsumgebung in Ihrem Namen mit derselben Ausführungsrolle durchführt, enthalten nicht die Quellfunktion ARN. Beispiele für solche API-Aufrufe außerhalb der Ausführungsumgebung sind:

- Ruft AWS Key Management Service (AWS KMS) auf, um Ihre Umgebungsvariablen automatisch zu verschlüsseln und zu entschlüsseln.

- Ruft Amazon Elastic Compute Cloud (Amazon EC2) auf, um Elastic Network Interfaces (ENIs) für eine VPC-enabled Funktion zu erstellen.
- Aufrufe AWS-Services, z. B. Amazon Simple Queue Service (Amazon SQS), um aus einer Ereignisquelle zu lesen, die als Ereignisquellen-Mapping eingerichtet ist.

Mit dem ARN der Quellfunktion im Kontext der Anmeldeinformationen können Sie überprüfen, ob ein Aufruf Ihrer Ressource aus dem Code einer bestimmten Lambda-Funktion stammt. Um dies zu überprüfen, verwenden Sie die `lambda:SourceFunctionArn`-Zustandsschlüssel in einer identitätsbasierten IAM-Richtlinie oder einer Service-Kontrollrichtlinie (SCP).

> **Note**
>
> Sie können den `lambda:SourceFunctionArn`-Bedingungsschlüssel nicht in ressourcenbasierten Richtlinien verwenden.

Mit diesem Bedingungsschlüssel in Ihren identitätsbasierten Richtlinien oder SCPs können Sie Sicherheitskontrollen für die API-Aktionen implementieren, die Ihr Funktionscode für andere AWS-Services vornimmt. Dies hat einige wichtige Sicherheitsanwendungen, z. B. um Ihnen zu helfen, die Quelle eines Anmeldeinformationslecks zu identifizieren.

> **Note**
>
> Der `lambda:SourceFunctionArn`-Bedingungsschlüssel unterscheidet sich von den `lambda:FunctionArn` und `aws:SourceArn`-Bedingungsschlüsseln. Der `lambda:FunctionArn`-Bedingungsschlüssel gilt nur für Ereignisquellenzuordnungen und hilft bei der Definition der Funktionen, die Ihre Ereignisquelle aufrufen kann. Der `aws:SourceArn` Bedingungsschlüssel gilt nur für Richtlinien, bei denen Ihre Lambda-Funktion die Zielressource ist, und hilft zu definieren, welche anderen AWS-Services AMD-Ressourcen diese Funktion aufrufen können. Der `lambda:SourceFunctionArn` Bedingungsschlüssel kann für jede identitätsbasierte Richtlinie oder jeden SCP gelten, um die spezifischen Lambda-Funktionen zu definieren, die berechtigt sind, bestimmte AWS API-Aufrufe an andere Ressourcen zu tätigen.

Um `lambda:SourceFunctionArn` in Ihrer Richtlinie zu verwenden, fügen Sie es als Bedingung mit einem der ARN-Bedingungsoperatoren ein. Der Wert des Schlüssels muss ein gültiger ARN sein.

Angenommen, Ihr Lambda-Funktionscode macht einen `s3:PutObject`-Aufruf, der auf einen bestimmten Amazon-S3-Bucket abzielt. Möglicherweise möchten Sie nur einer bestimmten Lambda-Funktion erlauben, dass `s3:PutObject` auf diesen Bucket zugreift. In diesem Fall sollte der Ausführungsrolle Ihrer Funktion eine Richtlinie angefügt sein, die wie folgt aussieht:

Example Richtlinie, die einer bestimmten Event-Funktion Zugriff auf eine Amazon S3 S3-Ressource gewährt

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

Diese Richtlinie erlaubt nur `s3:PutObject` Zugriff wenn die Quelle die Lambda-Funktion mit ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda` ist. Diese Richtlinie erlaubt `s3:PutObject` keinen Zugriff auf jede andere aufrufende Identität. Dies gilt auch dann, wenn eine andere Funktion oder Entität einen `s3:PutObject`-Aufruf mit der gleichen Ausführungsrolle tätigt.

> **Note**
>
> Der Bedingungsschlüssel `lambda:SourceFunctionARN` unterstützt keine Lambda-Funktionsversionen oder -Funktionsaliase. Wenn Sie den ARN für eine bestimmte Funktionsversion oder einen bestimmten Alias verwenden, ist Ihre Funktion nicht berechtigt,

die von Ihnen angegebene Aktion auszuführen. Achten Sie darauf, den unqualifizierten ARN für Ihre Funktion ohne Versions- oder Alias-Suffix zu verwenden.

Sie können `lambda:SourceFunctionArn` auch in SCPs verwenden. Angenommen, Sie möchten den Zugriff auf Ihren Bucket entweder auf den Code einer einzelnen Lambda-Funktion oder auf Aufrufe aus einer bestimmten Amazon Virtual Private Cloud (VPC) beschränken. Das folgende SCP illustriert dies.

Example Richtlinie, die den Zugriff auf Amazon S3 unter bestimmten Bedingungen verweigert

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                }
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
```

```
        }
    ]
}
```

Diese Richtlinie verweigert alle S3-Aktionen, sofern sie nicht von einer bestimmten Lambda-Funktion mit ARN `arn:aws:lambda:*:123456789012:function:source_lambda` stammen, oder sofern sie nicht aus dem angegebenen VPC stammen. Der `StringNotEqualsIfExists`-Operator weist IAM an, diese Bedingung nur zu verarbeiten, wenn der `aws:SourceVpc`-Schlüssel in der Anfrage vorhanden ist. In ähnlicher Weise berücksichtigt IAM den `ArnNotEqualsIfExists`-Operator nur, wenn `lambda:SourceFunctionArn` vorhanden ist.

Gewährung anderer AWS Entitäten Zugriff auf Ihre Event-Funktionen

Um anderen AWS-Konten Organisationen und Diensten Berechtigungen für den Zugriff auf Ihre Lambda-Ressourcen zu erteilen, haben Sie mehrere Möglichkeiten:

- Sie können identitätsbasierte Richtlinien verwenden, um anderen Benutzern Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Identity-based Richtlinien können direkt für Benutzer oder für Gruppen und Rollen gelten, die einem Benutzer zugeordnet sind.
- Sie können ressourcenbasierte Richtlinien verwenden, um anderen Konten und AWS-Services Berechtigungen für den Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Wenn ein Benutzer versucht, auf eine Lambda-Ressource zuzugreifen, berücksichtigt Lambda sowohl die identitätsbasierten Richtlinien des Benutzers als auch die ressourcenbasierte Richtlinie der Ressource. Wenn ein AWS Service wie Amazon Simple Storage Service (Amazon S3) Ihre Lambda-Funktion aufruft, berücksichtigt Lambda nur die ressourcenbasierte Richtlinie.
- Sie können ein attributbasiertes Zugriffskontrollmodell (ABAC) verwenden, um den Zugriff auf Ihre Lambda-Funktionen zu steuern. Mit ABAC können Sie Tags an eine Lambda-Funktion anhängen, sie in bestimmten API-Anforderungen übergeben oder sie an den IAM-Prinzipal anfügen, der die Anforderung stellt. Geben Sie dieselben Tags im Bedingungselement einer IAM-Richtlinie an, um den Funktionszugriff zu steuern.

Um Ihnen bei der Optimierung Ihrer Berechtigungen für den Zugriff mit den geringsten Rechten zu helfen, bietet Lambda einige zusätzliche Bedingungen, die Sie in Ihre Richtlinien aufnehmen können. Weitere Informationen finden Sie unter the section called “Ressourcen und Bedingungen”.

Identity-based IAM-Richtlinien für Lambda

Sie können identitätsbasierte Richtlinien in AWS Identity and Access Management (IAM) verwenden, um Benutzern in Ihrem Konto Zugriff auf Lambda zu gewähren. Identity-based Richtlinien können für Benutzer, Benutzergruppen oder Rollen gelten. Sie können auch Benutzern in einem anderen Konto die Berechtigung erteilen, eine Rolle in Ihrem Konto zu übernehmen und auf Ihre Lambda-Ressourcen zuzugreifen.

Lambda bietet AWS verwaltete Richtlinien, die Zugriff auf Lambda-API-Aktionen und in einigen Fällen Zugriff auf andere AWS Dienste gewähren, die zur Entwicklung und Verwaltung von Lambda-Ressourcen verwendet werden. Lambda aktualisiert je nach Bedarf die verwalteten Richtlinien, um sicherzustellen, dass Ihre Benutzer Zugriff auf neue Funktionen haben, sobald diese veröffentlicht werden.

- AWS Lambda_FullAccess — Gewährt vollen Zugriff auf Lambda-Aktionen und andere AWS Services, die zur Entwicklung und Wartung von Lambda-Ressourcen verwendet werden.
- AWS Lambda_ReadOnlyAccess — Gewährt schreibgeschützten Zugriff auf Lambda-Ressourcen.
- AWSLambdaRole— Erteilt Berechtigungen zum Aufrufen von Event-Funktionen.

AWS verwaltete Richtlinien gewähren Berechtigungen für API-Aktionen, ohne die Lambda-Funktionen oder -Layer einzuschränken, die ein Benutzer ändern kann. Für eine feinere Steuerung können Sie eigene Richtlinien erstellen, die den Umfang der Berechtigungen eines Benutzers einschränken. Informationen zur Gewährung von Layer-Zugriff finden Sie unter. the section called “Zugang zur Ebene”

Themen

- Benutzern Zugriff auf eine Event-Funktion gewähren
- Benutzern den Zugriff auf eine Lambda-Ebene gewähren

Benutzern Zugriff auf eine Event-Funktion gewähren

Verwenden Sie identitätsbasierte Richtlinien, um Benutzern, Benutzergruppen oder Rollen die Ausführung von Operationen mit Lambda Event Functions zu ermöglichen.

> **Note**
>
> Für eine als Container-Image definierte Funktion muss die Benutzerberechtigung für den Zugriff auf das Image in Amazon Elastic Container Registry (Amazon ECR) konfiguriert werden. Ein Beispiel finden Sie unter Amazon ECR-Repository-Richtlinien.

Im Folgenden finden Sie ein Beispiel für eine Berechtigungsrichtlinie mit eingeschränktem Umfang. Sie ermöglicht Benutzern das Erstellen und Verwalten von Lambda-Funktionen mit einem bestimmten Präfix (`intern-`), die mit einer bestimmten Ausführungsrolle konfiguriert sind.

Example Richtlinie für die Funktionsentwicklung

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "ArnLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
```

```
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
        }
    ]
}
```

Die Berechtigungen in der Richtlinie sind in Anweisungen basierend auf den Ressourcen und Bedingungen organisiert, die sie unterstützen.

- `ReadOnlyPermissions` – Die Lambda-Konsole verwendet diese Berechtigungen, wenn Sie Funktionen durchsuchen und anzeigen. Sie unterstützen keine Ressourcenmuster oder -bedingungen.

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda:GetEventSourceMapping",
            "lambda:GetFunction",
            "lambda:GetFunctionConfiguration",
            "lambda:GetFunctionCodeSigningConfig",
            "lambda:GetFunctionConcurrency",
            "lambda:ListEventSourceMappings",
            "lambda:ListFunctions",
            "lambda:ListTags",
            "iam:ListRoles"
        ],
```

```
            "Resource": "*"
```

- `DevelopFunctions` – Verwendet eine beliebige Lambda-Aktion, die auf Funktionen mit dem Präfix `intern-` ausgeführt wird, außer `AddPermission` und `PutFunctionConcurrency`. `AddPermission` ändert die ressourcenbasierte Richtlinie der Funktion und kann mögliche Sicherheitsprobleme darstellen. `PutFunctionConcurrency` reserviert Skalierungskapazitäten für eine Funktion und kann anderen Funktionen Kapazitäten wegnehmen.

```
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- `DevelopEventSourceMappings` – Verwaltet Ereignis-Quellzuweisungen für Funktionen mit dem Präfix `intern-`. Diese Aktionen werden für Ereignis-Quellzuweisungen angewendet. Sie können sie jedoch mit einer Bedingung nach Funktion einschränken.

```
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
```

- `PassExecutionRole` – Zeigt und übergibt nur eine Rolle namens `intern-lambda-execution-role`, die von einem Benutzer mit IAM-Berechtigungen erstellt und verwaltet werden muss. `PassRole` wird verwendet, wenn Sie eine Ausführungsrolle einer Funktion zuweisen.

```
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
```

```
            "iam:PassRole",
            "iam:SimulatePrincipalPolicy"
        ],
        "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs`— Verwenden Sie CloudWatch Logs, um Logs für Funktionen anzuzeigen, denen das Präfix vorangestellt ist. `intern-`

```
        "Action": [
            "logs:*"
        ],
        "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

Diese Richtlinie erlaubt es einem Benutzer, Lambda zu verwenden, ohne die Ressourcen anderer Benutzer zu gefährden. Es erlaubt einem Benutzer nicht, eine Funktion so zu konfigurieren, dass sie von anderen AWS Diensten ausgelöst wird oder diese aufruft, wofür umfassendere IAM-Berechtigungen erforderlich sind. Es beinhaltet auch keine Berechtigungen für Dienste, die keine Richtlinien mit begrenztem Geltungsbereich unterstützen, wie und. CloudWatch X-Ray Verwenden Sie die schreibgeschützten Richtlinien für diese Services, um dem Benutzer Zugriff auf Metriken und Ablaufverfolgungsdaten zu gewähren.

Wenn Sie Auslöser für Ihre Funktion konfigurieren, benötigen Sie Zugriff, um den AWS Dienst verwenden zu können, der Ihre Funktion aufruft. Wenn Sie z. B. einen Amazon-S3-Auslöser konfigurieren, müssen Sie zur Verwendung der Amazon-S3-Aktionen berechtigt sein, mit denen Bucket-Benachrichtigungen verwaltet werden. Viele dieser Berechtigungen sind in der von AWS Lambda_FullAccess verwalteten Richtlinie enthalten.

Benutzern den Zugriff auf eine Lambda-Ebene gewähren

Verwenden Sie identitätsbasierte Richtlinien, um Benutzern, Benutzergruppen oder Rollen die Durchführung von Vorgängen mit Lambda-Ebenen zu ermöglichen. Die folgende Richtlinie erteilt einem Benutzer die Berechtigung, Ebenen zu erstellen und sie mit Funktionen zu verwenden. Die Ressourcenmuster ermöglichen es dem Benutzer, in jeder AWS-Region beliebigen Layer-Version zu arbeiten, sofern der Name der Ebene mit beginnt`test-`.

Example Richtlinie für die Ebenenentwicklung

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
            "Action": [
                "lambda:PublishLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

Sie können die Ebenenverwendung auch bei der Erstellung und Konfiguration von Funktionen mit der `lambda:Layer`-Bedingung erzwingen. Sie können beispielsweise Benutzer daran hindern, Ebenen zu verwenden, die von anderen Konten veröffentlicht wurden. Die folgende Richtlinie fügt eine Bedingung zu den Aktionen `CreateFunction` und `UpdateFunctionConfiguration` hinzu, um zu verlangen, dass alle angegebenen Ebenen vom Konto `123456789012` stammen.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
```

```
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
}
```

Um sicherzustellen, dass die Bedingung angewendet wird, stellen Sie sicher, dass keine anderen Anweisungen dem Benutzer die Berechtigung für diese Aktionen erteilen.

Arbeiten mit ressourcenbasierten Richtlinien in Lambda

Mit ressourcenbasierten Berechtigungsrichtlinien können Sie anderen AWS-Konten Benutzern und Organisationen AWS-Services Zugriff auf Ihre Lambda Event Functions gewähren. Eine ressourcenbasierte Richtlinie ist ein JSON-Dokument, das eine oder mehrere Anweisungen enthält. Jede Anweisung definiert Folgendes:

- `Principal`: Die Entität, der Sie Berechtigungen erteilen möchten (eine andere AWS-Service, eine IAM-Rolle oder ein anderer Benutzer oder eine andere AWS-Konto)
- `Action`: Eine Liste der API-Aktionen, die Sie für den angegebenen Prinzipal zulassen oder verweigern möchten
- `Effect`: Ob Sie dem Prinzipal die Nutzung der ausgewählten API-Aktionen gestatten oder verweigern möchten
- `Resource`: Die Lambda-Funktion, die Version oder der Alias, für die die Anweisung gelten soll (Sie können auch ein Platzhalterzeichen verwenden, um alle Versionen und Aliase Ihrer Funktion anzugeben)

Sie können auch optionale Elemente wie `Sid` (eine Anweisungs-ID) und `Condition` (logische Bedingungen für eine detaillierte Zugriffskontrolle) verwenden. Eine vollständige Liste der unterstützten Richtlinienelemente finden Sie in der Referenz zu den IAM-JSON-Richtlinienelementen im AWS Identity and Access Management Benutzerhandbuch.

Hinzufügen ressourcenbasierter Berechtigungen zu einer Event-Funktion

Sie können Ihrer Lambda-Funktion mit zwei Methoden ressourcenbasierte Berechtigungen hinzufügen:

- Vollständige JSON-Richtlinie — Verwenden Sie die Lambda-Konsole oder die PutResourcePolicyAPI-Aktion AWS CLI, um ein vollständiges JSON-Richtliniendokument hinzuzufügen. Mit einer vollständigen JSON-Richtlinie können Sie die gesamte Palette der globalen IAM-Bedingungsschlüssel verwenden, mehrere Anweisungen mit mehreren Prinzipalen hinzufügen und explizite Anweisungen erstellen. `Deny` Die maximale Größe für eine auf JSON-Ressourcen basierende Richtlinie beträgt 20 KB.
- Individuelle Berechtigungen — Verwenden Sie die Konsole oder die AddPermissionAPI-Aktion, um einzelne `Allow` Anweisungen hinzuzufügen. Individuelle Berechtigungen unterstützen nur einen begrenzten Satz von Bedingungsschlüsseln (`aws:SourceArn`aws:SourceAccount, und`aws:PrincipalOrgID`).

Wir empfehlen Ihnen, vollständige JSON-Richtlinien zu definieren, um Ihrer Funktion ressourcenbasierte Berechtigungen hinzuzufügen. Die Erstellung einer vollständigen JSON-Richtlinie bietet Ihnen mehr Flexibilität und eine detaillierte Kontrolle über Ihre Berechtigungen.

> **Important**
>
> Durch die Verwendung werden alle vorhandenen ressourcenbasierten Richtlinien für die Ressource `put-resource-policy` ersetzt. Wenn für die Ressource bereits Berechtigungen definiert sind`add-permission`, werden diese `put-resource-policy` überschrieben. Dient `get-resource-policy` zum Abrufen der vorhandenen Richtlinie, bevor Sie Änderungen vornehmen.

Erforderliche Berechtigungen

Um die `DeleteResourcePolicy` API-Aktionen `PutResourcePolicy``GetResourcePolicy`, und verwenden zu können, benötigen Sie die folgenden IAM-Berechtigungen:

API-Aktion	Erforderliche Berechtigungen
PutResourcePolicy	`lambda:PutResourcePolicy` , `lambda:AddPermission` und `lambda:RemovePermission`
GetResourcePolicy	`lambda:GetResourcePolicy` und `lambda:GetPolicy`
DeleteResourcePolicy	`lambda:DeleteResourcePolicy` und `lambda:RemovePermission`

Console

Um eine vollständige, ressourcenbasierte JSON-Richtlinie zu erstellen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie Zugriff gewähren möchten, und wählen Sie dann die Registerkarte Konfiguration.
3. Wählen Sie Berechtigungen aus.
4. Wählen Sie im Bereich mit den Resource-based Richtlinienerklärungen die Option Bearbeiten aus. Diese Aktion öffnet den JSON-Richtlinieneditor.
5. Fügen Sie der Richtlinie Ihrer Funktion Anweisungen hinzu. Sie können API-Aktionen auswählen, Principals verschiedener Typen (Dienste, Konten, IAM-Rollen) hinzufügen und Bedingungsschlüssel hinzufügen, um den Zugriff zu kontrollieren.
6. Wählen Sie Speichern.

Sie können die ressourcenbasierte Richtlinie Ihrer Funktion auch direkt im Bereich Richtlinie bearbeiten.

AWS CLI

Um eine vollständige ressourcenbasierte JSON-Richtlinie hinzuzufügen

Verwenden Sie den Befehl, um Ihrer Funktion eine vollständige JSON-Richtlinie hinzuzufügen. `put-resource-policy` AWS CLI

Der folgende Beispielbefehl fügt Ihrer Funktion mithilfe einer Richtlinie, die in einer Datei mit dem Namen `policy.json` auf Ihrem lokalen Computer definiert ist, eine ressourcenbasierte Richtlinie hinzu. Führen Sie den Befehl in dem Verzeichnis aus, das die Datei enthält. Sie `resource-arn` können eine Funktionsversion oder einen Alias angeben, oder Sie können die unqualifizierte Funktion ARN verwenden, um die Richtlinie auf die gesamte Funktion anzuwenden.

```
aws lambda put-resource-policy --resource-arn arn:aws:lambda:us-
east-2:123456789012:function:my-function \
--policy file://policy.json
```

Lambda APIs

Verwenden Sie die PutResourcePolicyAPI-Aktion, um Ihrer Funktion eine vollständige JSON-Berechtigungsrichtlinie hinzuzufügen. Sie können die Richtlinie einer Funktion auch mithilfe der DeleteResourcePolicyAktion löschen oder mithilfe der GetResourcePolicyAktion die Richtlinie abrufen, die derzeit mit einer Funktion verknüpft ist.

Verwenden Sie die AddPermissionAPI-Aktion, um der Richtlinie einer Funktion individuelle Berechtigungen hinzuzufügen.

Die ressourcenbasierte Richtlinie einer Funktion anzeigen

Console

So zeigen Sie die ressourcenbasierte Richtlinie einer Funktion an

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Konfiguration und anschließend Berechtigungen aus.
4. Scrollen Sie nach unten zu den Resource-based Richtlinienerklärungen, um die Richtlinie einzusehen.

AWS CLI

Um die ressourcenbasierte Richtlinie einer Funktion anzuzeigen, verwenden Sie den Befehl `get-resource-policy`. Sie können den unqualifizierten ARN Ihrer Funktion verwenden oder eine Version oder einen Alias angeben.

```
aws lambda get-resource-policy --resource-arn arn:aws:lambda:us-
east-2:123456789012:function:my-function
```

Sie können auch den `get-policy` folgenden Befehl verwenden:

```
aws lambda get-policy \
  --function-name my-function \
  --output text
```

Lambda APIs

Verwenden Sie die GetResourcePolicyAPI-Aktion, um die ressourcenbasierte Richtlinie einer Funktion abzurufen. Sie können den unqualifizierten ARN Ihrer Funktion verwenden oder eine Version oder einen Alias-ARN angeben.

Sie können auch die GetPolicyAPI-Aktion verwenden.

Löschen der ressourcenbasierten Richtlinie einer Funktion

Console

Um die ressourcenbasierte Richtlinie einer Funktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie die Berechtigungsrichtlinie löschen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen.
4. Wählen Sie im Bereich mit den Resource-based Richtlinienerklärungen die Option Löschen aus.

AWS CLI

Um die Richtlinie Ihrer Funktion zu löschen, verwenden Sie den `delete-resource-policy` Befehl und geben Sie den ARN der Funktion, Version oder des Alias an, aus dem Sie die Richtlinie löschen möchten.

```
aws lambda delete-resource-policy --resource-arn arn:aws:lambda:us-
east-2:123456789012:function:my-function
```

Um einzelne Anweisungen zu entfernen, verwenden Sie`remove-permission`.

```
aws lambda remove-permission \
  --function-name example \
  --statement-id sns
```

Lambda APIs

Verwenden Sie die DeleteResourcePolicyAPI-Aktion, um die ressourcenbasierte Richtlinie zu löschen, die einer Funktion, Funktionsversion oder einem Funktionsalias zugeordnet ist.

Verwenden Sie die RemovePermissionAPI-Aktion, um einzelne Anweisungen aus einer Richtlinie zu entfernen.

Aktualisierung vorhandener Richtlinien

Wenn Sie die vorhandenen ressourcenbasierten Berechtigungen einer Funktion aktualisieren, hängt das Verhalten von der verwendeten Methode ab:

- `put-resource-policy`/PutResourcePolicy— Ersetzt die gesamte bestehende Richtlinie. Alle zuvor hinzugefügten individuellen Berechtigungen werden überschrieben.
- `add-permission`/AddPermission— Fügt der vorhandenen Richtlinie eine Anweisung hinzu, ohne sie zu überschreiben. Wenn Sie `add-permission` danach aufrufen`put-resource-policy`, wird die neue Anweisung an die bestehende JSON-Richtlinie angehängt.

Um zu vermeiden, dass bestehende Berechtigungen bei der Verwendung unbeabsichtigt überschrieben werden`put-resource-policy`, rufen Sie zuerst die bestehende Richtlinie Ihrer Funktion ab. Die `get-resource-policy` Ausgabe umfasst ein `RevisionId` Feld.

```
aws lambda get-resource-policy --resource-arn arn:aws:lambda:us-
east-2:123456789012:function:my-function
```

Wenn Sie eine neue Richtlinie anhängen, geben Sie den `RevisionId` Wert zusammen mit dem `--revision-id` Parameter an, um sicherzustellen, dass Sie die neueste Version aktualisieren. Wenn Sie eine ältere Revisions-ID angeben, aktualisiert Lambda die Richtlinie Ihrer Funktion nicht.

```
aws lambda put-resource-policy \
  --resource-arn arn:aws:lambda:us-east-2:123456789012:function:my-function \
  --policy file://policy.json \
  --revision-id a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
```

> **Note**
>
> Bestehende ressourcenbasierte Richtlinien, die mit erstellt wurden, funktionieren `AddPermission` weiterhin ohne Änderungen. Für die Verwendung der neuen `PutResourcePolicy` API sind keine Änderungen an Ihrem Funktionscode erforderlich.

Bewährte Methoden für die Gewährleistung der Sicherheit

Mit ressourcenbasierten JSON-Richtlinien können Sie den Zugriffsmustern mit den geringsten Rechten folgen. Sie können auch die gesetzlichen Anforderungen für explizite Verweigerungen erfüllen. Mit vollständigen JSON-Richtlinien können Sie:

- Erstellen Sie explizite `Deny` Anweisungen, um bestimmte Prinzipien oder Bedingungen zu blockieren.
- Verwenden Sie organisatorische Bedingungen (`aws:PrincipalOrgID`,`aws:PrincipalOrgPaths`), um den Zugriff auf Ihre Organizations einzuschränken, ohne einzelne Konten aufzuzählen.
- Geben Sie mithilfe der gesamten Palette der globalen IAM-Bedingungsschlüssel Zugriffsberechtigungen auf bestimmte Quellkonten oder ARNs ein.

Beispiel für ressourcenbasierte Richtlinien

Example Erteilen einer Genehmigung für Amazon S3 mit einer Ablehnungsaussage

Die folgende Richtlinie gewährt allen Amazon S3 S3-Buckets in einem Konto die Berechtigung, eine Funktion aufzurufen, mit Ausnahme eines Buckets, der ausdrücklich verweigert wurde.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-s3",
            "Effect": "Allow",
            "Principal": {
                "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
            "Condition": {
```

```
                "StringEquals": {
                    "aws:SourceAccount": "111122223333"
                }
            }
        },
        {
            "Sid": "deny-s3-bucket",
            "Effect": "Deny",
            "Principal": {
                "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": [
                "arn:aws:lambda:us-east-2:111122223333:function:my-function",
                "arn:aws:lambda:us-east-2:111122223333:function:my-function:*"
            ],
            "Condition": {
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:s3:::amzn-s3-demo-bucket"
                }
            }
        }
    ]
}
```

Example Erteilen von Berechtigungen für Konten in einer Organisation

Mit der folgenden Richtlinie wird allen Mitgliedern einer Organisation AWS-Konten der Zugriff gewährt.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "org-access",
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Principal": "*",
            "Resource": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
```

```
            }
        }
    ]
}
```

Example Erteilen von Berechtigungen für mehrere IAM-Rollen unter bestimmten Bedingungen

Die folgende Richtlinie gewährt zwei IAM-Rollen die Erlaubnis, die CreateAlias Aktion von einer bestimmten IP-Adresse aus zu verwenden.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-roles",
            "Effect": "Allow",
            "Action": "lambda:CreateAlias",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::444455556666:role/role-name",
                    "arn:aws:iam::444455556666:role/role-name2"
                ]
            },
            "Resource": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": "192.0.2.0"
                }
            }
        }
    ]
}
```

Example Der Zugriff wird verweigert, sofern der Anrufer nicht Teil einer bestimmten Organisation ist

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "deny-access",
            "Effect": "Deny",
            "Action": "lambda:InvokeFunction",
            "Principal": "*",
```

```
            "Resource": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
            "Condition": {
                "ForAllValues:StringNotLike": {
                    "aws:PrincipalOrgPaths": [
                        "o-a1b2c3d4e5/r-ab12/ou-ab12-11111111/*"
                    ]
                }
            }
        },
        {
            "Sid": "allow-access",
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Principal": "*",
            "Resource": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
            "Condition": {
                "ForAnyValue:StringLike": {
                    "aws:PrincipalOrgPaths": [
                        "o-a1b2c3d4e5/r-ab12/ou-ab12-11111111/*"
                    ]
                }
            }
        }
    ]
}
```

Unterstützte API-Aktionen

Die folgenden Lambda-API-Aktionen unterstützen ressourcenbasierte Richtlinien:

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration

- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- InvokeFunctionUrl(Nur mit Genehmigung)
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
- UpdateFunctionEventInvokeConfig

Zugriff auf die Event-Funktion gewähren für AWS-Services

Wenn Sie einen AWS Dienst verwenden, um Ihre Funktion aufzurufen, erteilen Sie die Erlaubnis in einer Erklärung zu einer ressourcenbasierten Richtlinie. Sie können die Anweisung auf die gesamte Funktion anwenden oder die Anweisung auf eine einzelne Version oder einen einzelnen Alias beschränken.

> **Note**
>
> Wenn Sie Ihrer Funktion mit der Lambda-Konsole einen Auslöser hinzufügen, aktualisiert die Konsole die ressourcenbasierte Richtlinie der Funktion, damit der Service sie aufrufen kann. Um Berechtigungen für andere Konten oder Services zu erteilen, die in der Lambda-Konsole nicht verfügbar sind, verwenden Sie die AWS CLI-CLI.

Wir empfehlen die Verwendung `put-resource-policy` zur Definition einer vollständigen JSON-Richtlinie. Mit `put-resource-policy` können Sie Deny-Anweisungen hinzufügen, die gesamte Palette der IAM-Bedingungsschlüssel verwenden und alle Berechtigungen in einem einzigen Dokument verwalten. Sie können es auch verwenden`add-permission`, um einzelne Anweisungen für einfache Anwendungsfälle hinzuzufügen.

> **Important**
>
> Durch `put-resource-policy` die Verwendung werden alle vorhandenen ressourcenbasierten Richtlinien für die Ressource ersetzt. Wenn für die Ressource bereits Berechtigungen mit definiert wurden`add-permission`, werden diese `put-resource-policy` überschrieben. Dient `get-resource-policy` zum Abrufen der vorhandenen Richtlinie, bevor Sie Änderungen vornehmen.

Verwendung einer vollständigen JSON-Richtlinie (empfohlen)

Die folgende Beispielrichtlinie gewährt Amazon Simple Notification Service die Erlaubnis, eine Funktion mit dem Namen aufzurufen`my-function`, jedoch nur für ein bestimmtes Thema. Außerdem werden Aufrufe aus einem bestimmten Thema verweigert.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-sns-my-topic",
            "Effect": "Allow",
            "Principal": {
                "Service": "sns.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
            "Condition": {
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:sns:us-east-2:123456789012:my-topic"
                }
            }
        },
        {
            "Sid": "deny-sns-other-topic",
            "Effect": "Deny",
```

```
            "Principal": {
                "Service": "sns.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": [
                "arn:aws:lambda:us-east-2:123456789012:function:my-function",
                "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ],
            "Condition": {
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:sns:us-east-2:123456789012:restricted-
topic"
                }
            }
        }
    ]
}
```

Speichern Sie die Richtlinie in einer Datei mit dem Namen `policy.json` und wenden Sie sie an mit: `put-resource-policy`

```
aws lambda put-resource-policy \
  --resource-arn arn:aws:lambda:us-east-2:123456789012:function:my-function \
  --policy file://policy.json
```

Für Amazon S3 ist die Quelle ein Bucket, dessen ARN keine Konto-ID enthält. Es ist möglich, dass Sie einen Bucket löschen und ein anderes Konto einen Bucket mit demselben Namen erstellen könnte. Verwenden Sie den `aws:SourceAccount` Bedingungsschlüssel, um sicherzustellen, dass nur Ressourcen in Ihrem Konto die Funktion aufrufen können:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-s3",
            "Effect": "Allow",
            "Principal": {
                "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
            "Condition": {
```

```
                "StringEquals": {
                    "aws:SourceAccount": "123456789012"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:s3:::my-bucket"
                }
            }
        }
    ]
}
```

Verwenden von add-permission

Für einfache Anwendungsfälle können Sie verwenden, `add-permission` um einzelne Anweisungen hinzuzufügen. Der folgende Befehl erteilt Amazon Simple Notification Service die Erlaubnis, eine Funktion mit dem Namen `my-function` aufzurufen:

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns \
  --principal sns.amazonaws.com \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

> **Note**
>
> Wenn Sie `add-permission` danach aufrufen`put-resource-policy`, wird die neue Anweisung an die bestehende JSON-Richtlinie angehängt.

Um Aufrufe auf eine bestimmte Ressource zu beschränken, verwenden Sie die Option: `source-arn`

```
aws lambda add-permission \
  --function-name my-function \
```

```
  --action lambda:InvokeFunction \
  --statement-id sns-my-topic \
  --principal sns.amazonaws.com \
  --source-arn arn:aws:sns:us-east-2:123456789012:my-topic
```

Verwenden Sie für Amazon S3 die `source-account` Option mit Ihrer Konto-ID:

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id s3-account \
  --principal s3.amazonaws.com \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

Gewähren des Funktionszugriffs auf eine Organisation

Um allen Konten in einer AWS OrganizationsOrganisation Berechtigungen zu erteilen, empfehlen wir, eine vollständige JSON-Richtlinie mit dem `aws:PrincipalOrgID` Bedingungsschlüssel `put-resource-policy` zu erstellen.

> **Important**
>
> Durch `put-resource-policy` die Verwendung werden alle vorhandenen ressourcenbasierten Richtlinien für die Ressource ersetzt. Wenn für die Ressource bereits Berechtigungen mit definiert wurden`add-permission`, werden diese `put-resource-policy` überschrieben. Dient `get-resource-policy` zum Abrufen der vorhandenen Richtlinie, bevor Sie Änderungen vornehmen.

Verwendung einer vollständigen JSON-Richtlinie (empfohlen)

Die folgende Beispielrichtlinie gewährt allen AWS-Konten Mitgliedern der Organisation `o-a1b2c3d4e5f` Aufrufzugriff:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "org-access",
```

```
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

Speichern Sie die Richtlinie in einer Datei mit dem Namen `policy.json` und wenden Sie sie an:

```
aws lambda put-resource-policy \
  --resource-arn arn:aws:lambda:us-east-2:123456789012:function:my-function \
  --policy file://policy.json
```

Mit einer vollständigen JSON-Richtlinie können Sie auch den Zugriff auf bestimmte Konten innerhalb einer Organisation verweigern:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-org",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        },
        {
            "Sid": "deny-specific-account",
            "Effect": "Deny",
            "Principal": {
                "AWS": "arn:aws:iam::999888777666:root"
            },
```

```
            "Action": "lambda:InvokeFunction",
            "Resource": [
                "arn:aws:lambda:us-east-2:123456789012:function:my-function",
                "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
        }
    ]
}
```

Verwenden von Add-Permission

Für einfache Anwendungsfälle können Sie die `principal-org-id` Option verwenden`add-permission`. Der folgende Befehl gewährt allen Benutzern in der Organisation `o-a1b2c3d4e5f` Aufrufzugriff:

```
aws lambda add-permission \
  --function-name my-function \
  --statement-id org-access \
  --action lambda:InvokeFunction \
  --principal * \
  --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> In diesem Befehl ist `Principal *`. Dies bedeutet, dass alle Benutzer in der Organisation `o-a1b2c3d4e5f` Berechtigungen für den Funktionsaufruf erhalten. Wenn Sie eine AWS-Konto Oder-Rolle als angeben`Principal`, erhält nur dieser Prinzipal die Berechtigungen zum Aufrufen von Funktionen, aber nur, wenn er auch Teil der `o-a1b2c3d4e5f` Organisation ist.

Weitere Informationen finden Sie unter aws: PrincipalOrg ID im IAM-Benutzerhandbuch.

Zugriff auf Event Function für andere Konten gewähren

Um eine Funktion mit einer anderen gemeinsam zu nutzen AWS-Konto, fügen Sie der ressourcenbasierten Richtlinie der Funktion eine kontoübergreifende Berechtigungsanweisung hinzu. Wir empfehlen die Verwendung`put-resource-policy`, um eine vollständige JSON-Richtlinie zu erstellen. Mit `put-resource-policy` können Sie mehrere Principals angeben, Bedingungen hinzufügen und Ablehnungsbefehle erstellen. Sie können es auch `add-permission` für einfache Anwendungsfälle verwenden.

> **Important**
>
> Durch `put-resource-policy` die Verwendung werden alle vorhandenen ressourcenbasierten Richtlinien für die Ressource ersetzt. Wenn für die Ressource bereits Berechtigungen mit definiert wurden`add-permission`, werden diese `put-resource-policy` überschrieben. Dient `get-resource-policy` zum Abrufen der vorhandenen Richtlinie, bevor Sie Änderungen vornehmen.

Verwendung einer vollständigen JSON-Richtlinie (empfohlen)

Die folgende Beispielrichtlinie gewährt zwei AWS-Konten Berechtigungen zum Aufrufen einer Funktion, wobei eine auf eine bestimmte IAM-Rolle beschränkt ist:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-account-invoke",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:root",
                    "arn:aws:iam::444455556666:role/cross-account-role"
                ]
            },
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function"
        }
    ]
}
```

Speichern Sie die Richtlinie in einer Datei mit dem Namen `policy.json` und wenden Sie sie an. Sie `resource-arn` können eine Funktion, eine Version oder einen Alias angeben:

```
aws lambda put-resource-policy \
  --resource-arn arn:aws:lambda:us-east-2:123456789012:function:my-function \
  --policy file://policy.json
```

Sie können den kontoübergreifenden Zugriff auch mithilfe eines Alias auf bestimmte Versionen einschränken. Die folgende Richtlinie gewährt Aufrufzugriff nur über den `prod` Alias:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "allow-account-prod-alias",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:prod"
        }
    ]
}
```

Speichern Sie diese Richtlinie in einer Datei mit dem Namen `alias-policy.json` und wenden Sie sie auf den Alias an:

```
aws lambda put-resource-policy \
  --resource-arn arn:aws:lambda:us-east-2:123456789012:function:my-function:prod \
  --policy file://alias-policy.json
```

Die ressourcenbasierte Richtlinie erteilt dem anderen Konto die Berechtigung für den Zugriff auf die Funktion, erlaubt den Benutzern in diesem Konto jedoch nicht, ihre Berechtigungen zu überschreiten. Benutzer in dem anderen Konto müssen über die entsprechenden Benutzerberechtigungen verfügen, um die Lambda-API verwenden zu können.

Sie können kontoübergreifenden Zugriff für die meisten API-Aktionen erteilen, die für eine vorhandene Funktion ausgeführt werden. Beispielsweise könnten Sie Zugriff auf `lambda:ListAliases` gewähren, damit ein Konto eine Liste der Aliasse abrufen kann, oder `lambda:GetFunction`, damit es Ihren Funktionscode herunterladen kann.

Um anderen Konten Berechtigungen für mehrere Funktionen oder für Aktionen zu erteilen, die nicht für eine Funktion ausgeführt werden, empfehlen wir die Verwendung von IAM-Rollen.

Verwenden von add-permission

Für einfache Anwendungsfälle können Sie verwenden, `add-permission` um einzelne Anweisungen hinzuzufügen. Im folgenden Beispiel wird dem Konto die `111122223333` Berechtigung zum Aufrufen `my-function` mit dem `prod` Alias erteilt:

```
aws lambda add-permission \
  --function-name my-function:prod \
  --statement-id xaccount \
  --action lambda:InvokeFunction \
  --principal 111122223333 \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-1:123456789012:function:my-function"}
```

> **Note**
>
> Wenn Sie `add-permission` danach aufrufen`put-resource-policy`, wird die neue Anweisung an die bestehende JSON-Richtlinie angehängt.

Um den Zugriff auf einen Benutzer oder eine Rolle in einem anderen Konto einzuschränken, geben Sie den vollständigen ARN der Identität als Prinzipal an. Beispiel, `arn:aws:iam::123456789012:user/developer`.

Gewähren des Lambda-Ebenenzugriffs für andere Konten

Um eine Ebene mit einer anderen gemeinsam zu nutzen AWS-Konto, fügen Sie der ressourcenbasierten Richtlinie der Ebene eine kontoübergreifende Berechtigungserklärung hinzu. Führen Sie den add-layer-version-permissionBefehl aus und geben Sie die Konto-ID als. `principal` In jeder Anweisung können Sie einem einzelnen Konto, allen Konten oder einer Organisation in AWS Organizations eine Berechtigung erteilen.

Im folgenden Beispiel wird dem Konto 111122223333 Zugriff auf Version 2 der `bash-runtime`-Ebene gewährt.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
```

```
  --principal 111122223333 \
  --output text
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Berechtigungen gelten nur für eine Version mit einer Ebene. Wiederholen Sie den Vorgang bei jeder Erstellung einer neuen Ebenenversion.

Um Berechtigungen für alle Konten in einer AWS Organizations Organisation zu erteilen, verwenden Sie die Option `organization-id`. Im folgenden Beispiel wird allen Konten in Organisation `o-t194hfs8cz` die Berechtigung für die Verwendung von Version 3 einer `my-layer` erteilt.

```
aws lambda add-layer-version-permission \
  --layer-name my-layer \
  --version-number 3 \
  --statement-id engineering-org \
  --principal '*' \
  --action lambda:GetLayerVersion \
  --organization-id o-t194hfs8cz \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

Um die Berechtigung für mehrere Konten oder Organisationen zu erteilen, müssen Sie mehrere Bescheinigungen hinzufügen.

Verwendung der attributbasierten Zugriffskontrolle in Lambda

Mit attributbasierter Zugriffskontrolle (ABAC) können Sie Tags verwenden, um den Zugriff auf Ihre Lambda-Ressourcen zu steuern. Sie können Tags an bestimmte Lambda-Ressourcen anhängen, sie an bestimmte API-Anfragen anhängen oder sie an den AWS Identity and Access Management (IAM-)

Principal anhängen, der die Anfrage stellt. Weitere Informationen darüber, wie attributebasierten Zugriff AWS gewährt wird, finden Sie im IAM-Benutzerhandbuch unter Steuern des Zugriffs auf AWS Ressourcen mithilfe von Tags.

Sie können ABAC verwenden, um die geringsten Berechtigungen zu gewähren, ohne einen Amazon-Ressourcennamen (ARN) oder ein ARN-Muster in der IAM-Richtlinie anzugeben. Stattdessen können Sie ein Tag im Bedingungselement einer IAM-Richtlinie angeben, um den Zugriff zu steuern. Die Skalierung ist mit ABAC einfacher, da Sie Ihre IAM-Richtlinien nicht aktualisieren müssen, wenn Sie neue Ressourcen erstellen. Fügen Sie stattdessen Tags zu den neuen Ressourcen hinzu, um den Zugriff zu steuern.

In Lambda funktionieren Tags für die folgenden Ressourcen:

- Funktionen – Weitere Informationen zu Tagging-Funktionen finden Sie unter the section called “Tags (Markierungen)”.
- Codesignaturkonfigurationen – Weitere Informationen zum Markieren von Codesignaturkonfigurationen finden Sie unter the section called “Code-Signaturkonfigurations-Tags”.
- Zuordnungen von Ereignisquellen – Weitere Informationen zum Markieren von Zuordnungen von Ereignisquellen finden Sie unter the section called “Tags für die Zuordnung von Ereignisquellen”.

Tags werden für Ebenen nicht unterstützt.

Sie können die folgenden Bedingungsschlüssel verwenden, um auf Tag basierende IAM-Richtlinienregeln zu schreiben:

- aws:ResourceTag/tag-key: Steuern Sie den Zugriff anhand der Tags, die an eine Lambda-Ressource angehängt sind.
- aws: RequestTag/tag-key: Erfordert, dass Tags in einer Anfrage vorhanden sind, z. B. beim Erstellen einer neuen Funktion.
- aws:PrincipalTag/tag-key: Steuern Sie anhand der Tags, die ihrem IAM-Benutzer oder ihrer IAM-Rolle zugewiesen sind, was der IAM-Prinzipal (die Person, die die Anfrage stellt) tun darf.
- aws:TagKeys: Steuert, ob bestimmte Tag-Schlüssel in einer Anfrage verwendet werden können.

Sie können nur Bedingungen für Aktionen angeben, die diese Bedingungen unterstützen. Eine Liste der Bedingungen, die von jeder Lambda-Aktion unterstützt werden, finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Lambda in der Referenz zur Dienstautorisierung.

Informationen zur ResourceTag/tag-key Unterstützung von aws: finden Sie unter „Ressourcentypen definiert von“ AWS Lambda. Informationen zum TagKeys Support von aws: RequestTag/tag-key und aws: finden Sie unter „Aktionen definiert von“ AWS Lambda.

Themen

- Sichern Sie Ihre Funktionen per Tag

Sichern Sie Ihre Funktionen per Tag

Die folgenden Schritte zeigen eine Möglichkeit, wie Sie mit ABAC Berechtigungen für Funktionen einrichten können. In diesem Beispielszenario erstellen Sie vier IAM-Berechtigungsrichtlinien. Anschließend fügen Sie diese Richtlinien einer neuen IAM-Rolle hinzu. Schließlich erstellen Sie einen IAM-Benutzer und erteilen diesem Benutzer die Berechtigung, die neue Rolle zu übernehmen.

Themen

- Voraussetzungen
- Schritt 1: Tags für neue Funktionen anfordern
- Schritt 2: Lassen Sie Aktionen zu, die auf Tags basieren, die einer Event-Funktion und einem IAM-Principal zugeordnet sind
- Schritt 3: Erteilen von Listenberechtigungen
- Schritt 4: Erteilen von IAM-Berechtigungen
- Schritt 5: Erstellen der IAM-Rolle
- Schritt 6: Erstellen eines IAM-Benutzers
- Schritt 7: Testen der Berechtigungen
- Schritt 8: Bereinigen Ihrer Ressourcen

Voraussetzungen

Stellen Sie sicher, dass Sie über eine Lambda-Ausführungsrolle verfügen. Sie verwenden diese Rolle, wenn Sie IAM-Berechtigungen erteilen und wenn Sie eine Event-Funktion erstellen.

Schritt 1: Tags für neue Funktionen anfordern

Wenn Sie ABAC mit Lambda verwenden, empfiehlt es sich, dass alle Funktionen über Tags verfügen. Auf diese Weise können Sie sicherstellen, dass Ihre ABAC-Berechtigungsrichtlinien wie erwartet funktionieren.

Erstellen Sie eine IAM-Richtlinie ähnlich dem folgenden Beispiel. Diese Richtlinie verwendet die TagKeys Bedingungsschlüssel aws: RequestTag/tag-key, aws: und aws: ResourceTag/tag-key, um zu verlangen, dass neue Funktionen und der IAM-Prinzipal, der die Funktionen erstellt, beide das `project` Tag haben. Der `ForAllValues`-Modifikator stellt sicher, dass `project` das einzige zulässige Tag ist. Wenn Sie den `ForAllValues`-Modifikator nicht angeben, können die Benutzer der Funktion andere Tags hinzufügen, solange diese auch `project` übergeben.

Example– Tags für neue Funktionen anfordern

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:TagResource"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/project": "${aws:PrincipalTag/project}",
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        },
        "ForAllValues:StringEquals": {
          "aws:TagKeys": "project"
        }
      }
    }
  }
```

Schritt 2: Lassen Sie Aktionen zu, die auf Tags basieren, die einer Event-Funktion und einem IAM-Principal zugeordnet sind

Erstellen Sie eine zweite IAM-Richtlinie mit dem ResourceTag/tag-key Bedingungsschlüssel aws:, sodass das Tag des Prinzipals mit dem Tag übereinstimmen muss, das der Funktion zugeordnet ist. Die folgende Beispielrichtlinie erlaubt es Prinzipalen mit dem `project`-Tag, Funktionen mit dem `project`-Tag aufzurufen. Wenn eine Funktion andere Tags hat, wird die Aktion verweigert.

Example– Erfordert übereinstimmende Tags für Funktion und IAM-Prinzipal

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "lambda:InvokeFunction",
          "lambda:GetFunction"
        ],
        "Resource": "arn:aws:lambda:*:*:function:*",
        "Condition": {
          "StringEquals": {
            "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
          }
        }
      }
    ]
  }
```

Schritt 3: Erteilen von Listenberechtigungen

Erstellen Sie eine Richtlinie, die es dem Prinzipal ermöglicht, Lambda-Funktionen und IAM-Rollen aufzulisten. Dadurch kann der Prinzipal alle Lambda-Funktionen und IAM-Rollen auf der Konsole und beim Aufrufen der API-Aktionen sehen.

Example– Erteilen von Lambda- und IAM-Listenberechtigungen

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "AllResourcesLambdaNoTags",
        "Effect": "Allow",
        "Action": [
```

```
            "lambda:GetAccountSettings",
            "lambda:ListFunctions",
            "iam:ListRoles"
        ],
        "Resource": "*"
      }
    ]
  }
```

Schritt 4: Erteilen von IAM-Berechtigungen

Erstellen Sie eine Richtlinie, die iam: ermöglicht. PassRole Diese Berechtigung ist erforderlich, wenn Sie einer Funktion eine Ausführungsrolle zuweisen. Ersetzen Sie in der folgenden Beispielrichtlinie den Beispiel-ARN durch den ARN Ihrer Lambda-Ausführungsrolle.

> **Note**
>
> Verwenden Sie den `ResourceTag`-Bedingungsschlüssel nicht in einer Richtlinie mit der `iam:PassRole`-Aktion. Sie können das Tag nicht für eine IAM-Rolle verwenden, um zu steuern, wer Zugriff zur Weiterleitung diese Rolle haben soll. Weitere Informationen zu den Berechtigungen, die für die Übergabe einer Rolle an einen Dienst erforderlich sind, finden Sie unter Einem Benutzer Berechtigungen zur Übergabe einer Rolle an einen AWS Dienst gewähren.

Example– Erteilt die Berechtigung zum Übergeben der Ausführungsrolle

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
```

```
      }
    ]
  }
```

Schritt 5: Erstellen der IAM-Rolle

Es hat sich bewährt, Rollen zum Delegieren von Berechtigungen zu verwenden. Erstellen Sie eine IAM-Rolle mit dem Namen `abac-project-role`:

- In Schritt 1: Vertrauenswürdige Entität auswählen: Wählen Sie wählen AWS -Konto und wählen Sie dann Dieses Konto.
- In Schritt 2: Hinzufügen von Berechtigungen: Fügen Sie die vier IAM-Richtlinien hinzu, die Sie in den vorherigen Schritten erstellt haben.
- In Schritt 3: Benennen, Überprüfen und Erstellen: Wählen Sie Tag hinzufügen. Geben Sie für Key (Schlüssel) `project` ein. Geben Sie keinen Value (Wert) ein.

Schritt 6: Erstellen eines IAM-Benutzers

Erstellen Sie einen IAM-Benutzer mit dem Namen `abac-test-user`. Wählen Sie im Abschnitt Set permissions (Berechtigungen festlegen) die Option Attach existing policies directly (Vorhandene Richtlinien direkt anhängen) und dann Create policy (Richtlinie erstellen) aus. Geben Sie die folgende Richtliniendefinition ein. Ersetzen Sie *111122223333* durch Ihre AWS -Konto-ID. Diese Richtlinie erlaubt es `abac-test-user`, `abac-project-role` anzunehmen.

Example– IAM-Benutzer die Übernahme der ABAC-Rolle erlauben

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
    }
  }
```

Schritt 7: Testen der Berechtigungen

1. Melden Sie sich an der AWS Konsole an als `abac-test-user`. Weitere Informationen finden Sie unter Anmeldung als IAM-Benutzer.
2. Wechseln Sie zur `abac-project-role`-Rolle. Weitere Informationen finden Sie unter Wechseln zu einer Rolle (Konsole).
3. Erstellen einer Lambda-Funktion:
   - Wählen Sie unter Permissions (Berechtigungen) die Option Change default execution role (Standardausführungsrolle ändern) und dann unter Execution role (Ausführungsrolle) die Option Use an existing role (Vorhandene Rolle verwenden). Wählen Sie dieselbe Ausführungsrolle aus, die Sie in Schritt 4: Erteilen von IAM-Berechtigungen verwendet haben.
   - Wählen Sie unter Advanced settings (Erweiterte Einstellungen) die Option Enable tags (Tags ermöglichen) und wählen Sie dann Add new tag (Neues Tag hinzufügen). Geben Sie für Key (Schlüssel) `project` ein. Geben Sie keinen Value (Wert) ein.
4. Testen der Funktion.
5. Erstellen Sie eine zweite Lambda-Funktion und fügen Sie ein anderes Tag hinzu, z. B. `environment`. Dieser Vorgang sollte fehlschlagen, da die ABAC-Richtlinie, die Sie in Schritt 1: Tags für neue Funktionen anfordern erstellt haben, dem Prinzipal nur erlaubt, Funktionen mit dem `project`-Tag zu erstellen.
6. Erstellen Sie eine dritte Funktion ohne Tags. Dieser Vorgang sollte fehlschlagen, weil die ABAC-Richtlinie, die Sie in Schritt 1: Tags für neue Funktionen anfordern erstellt haben, dem Prinzipal nicht erlaubt, Funktionen ohne Tags zu erstellen.

Mit dieser Autorisierungsstrategie können Sie den Zugriff kontrollieren, ohne für jeden neuen Benutzer neue Richtlinien erstellen zu müssen. Um neuen Benutzern Zugriff zu gewähren, erteilen Sie ihnen einfach die Berechtigung, die Rolle zu übernehmen, die ihrem zugewiesenen Projekt entspricht.

Schritt 8: Bereinigen Ihrer Ressourcen

Löschen Sie die IAM-Rolle wie folgt:

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie die Rolle aus, die Sie im Schritt 5 erstellt haben.
3. Wählen Sie Löschen aus.

4. Geben Sie den Namen der Rolle in das Texteingabefeld ein, um den Löschvorgang zu bestätigen.
5. Wählen Sie Löschen aus.

So löschen Sie den IAM-Benutzer

1. Öffnen Sie die Seite Users in der IAM-Konsole.
2. Wählen Sie den IAM-Benutzer aus, den Sie in Schritt 6 erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen des Benutzers in das Texteingabefeld ein, um den Löschvorgang zu bestätigen.
5. Wählen Sie Benutzer löschen.

Um die Lambda-Ereignisfunktion zu löschen

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Ereignisfunktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Fine-tuning die Abschnitte „Ressourcen“ und „Bedingungen“ der Richtlinien

Sie können den Umfang der Berechtigungen eines Benutzers einschränken, indem Sie Ressourcen und Bedingungen in einer AWS Identity and Access Management (IAM)-Richtlinie angeben. Jede Aktion in einer Richtlinie unterstützt eine Kombination aus Ressourcen- und Bedingungstypen, die je nach Verhalten der Aktion variiert.

Jede IAM-Richtlinienanweisung erteilt die Berechtigung für eine Aktion, die auf eine Ressource ausgeführt wird. Wenn die Aktion nicht auf eine benannte Ressource reagiert oder Sie die Berechtigung zum Ausführen der Aktion für alle Ressourcen erteilen, wird der Wert der Ressource in der Richtlinie als Platzhalter () dargeste (*). Für viele Aktionen können Sie die Ressourcen einschränken, die ein Benutzer ändern kann, indem Sie den Amazon-Ressourcennamen (ARN) einer Ressource oder ein ARN-Muster angeben, das mehreren Ressourcen entspricht.

Je nach Ressourcentyp lässt sich der Umfang einer Maßnahme generell wie folgt einschränken:

- Funktionen–Aktionen, die auf eine Funktion wirken, können auf eine bestimmte Funktion nach Funktion, Version oder Alias-ARN beschränkt werden.
- Zuordnungen von Ereignisquellen – Aktionen können per ARN auf bestimmte Ressourcen zur Zuordnung von Ereignisquellen beschränkt werden. Zuordnungen von Ereignisquellen sind immer mit einer Funktion verknüpft. Sie können auch die `lambda:FunctionArn`-Bedingung verwenden, um Aktionen durch die zugehörige Funktion einzuschränken.
- Ebenen – Aktionen, die sich auf die Verwendung von Ebenen und Berechtigungen beziehen, wirken sich auf eine Version einer Ebene aus.
- Codesignaturkonfiguration – Aktionen können per ARN auf bestimmte Codesignatur-Konfigurationsressourcen beschränkt werden.
- Tags – Verwenden Sie standardmäßige Tag-Bedingungen. Weitere Informationen finden Sie unter the section called “Attribute-based Zugriffskontrolle”.

Zum Einschränken von Berechtigungen nach Ressource bestimmen Sie die Ressource unter Angabe des ARN.

Format des Lambda-Ressourcen-ARN

- Funktion – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*
- Funktionsversion – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`1`*
- Funktionsalias – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`TEST`*
- Dauerhafte Ausführung — `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`1`*`/durable-execution/`*`execution-name`*`/`*`execution-id`*
- Mapping von Ereignisquellen – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:event-source-mapping:`*`fa123456-14a1-4fd2-9fec-83de64ad683de6d47`*
- Ebene – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:layer:`*`my-layer`*
- Ebenenversion – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:layer:`*`my-layer`*`:`*`1`*
- Codesignaturkonfiguration – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:code-signing-config:`*`my-csc`*

Die folgende Richtlinie ermöglicht es einem Benutzer beispielsweise, eine Funktion AWS-Konto `123456789012` aufzurufen, die `my-function` in der AWS Region USA West (Oregon) benannt ist.

Example Aufrufen der Funktionsrichtlinie

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-
function"
        }
    ]
}
```

Dies ist ein Sonderfall, da sich die Aktionskennung (`lambda:InvokeFunction`) von der API-Operation (Invoke) unterscheidet. Bei anderen Aktionen ist die Aktionskennung der Operationsname mit dem Präfix `lambda:`.

Sections

- Den Abschnitt „Bedingungen“ in Richtlinien verstehen
- Referenzierung von Funktionen im Abschnitt „Ressourcen“ von Richtlinien
- Im Abschnitt „Ressourcen“ der Richtlinien wird auf dauerhafte Ausführungen verwiesen
- Unterstützte IAM-Aktionen und Funktionsverhalten

Den Abschnitt „Bedingungen“ in Richtlinien verstehen

Bedingungen sind ein optionales Richtlinienelement, das zusätzliche Logik anwendet, um zu bestimmen, ob eine Aktion zulässig ist. Zusätzlich zu gemeinsamen Bedingungen, die alle Aktionen unterstützen, definiert Lambda Bedingungstypen, die Sie verwenden können, um die Werte zusätzlicher Parameter für einige Aktionen einzuschränken.

Mit der Bedingung `lambda:Principal` können Sie beispielsweise den Service oder das Konto beschränken, für das ein Benutzer in der ressourcenbasierten Richtlinie einer Funktion Aufrufzugriff

gewähren kann. Mit der folgenden Richtlinie kann ein Benutzer Amazon Simple Notification Service (Amazon SNS)-Themen die Berechtigung erteilen, eine Funktion mit dem Namen `test` aufzurufen.

Example Verwalten von Berechtigungen für Funktionsrichtlinien

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

Die Bedingung erfordert, dass der Prinzipal Amazon SNS und kein anderer Service bzw. kein anderes Konto ist. Das Ressourcenmuster erfordert, dass der Funktionsname `test` lautet und eine Versionsnummer oder einen Alias enthält. Beispiel, `test:v1`.

Weitere Informationen zu Ressourcen und Bedingungen für Lambda und andere AWS Dienste finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Dienste in der Service Authorization Reference.

Referenzierung von Funktionen im Abschnitt „Ressourcen“ von Richtlinien

Sie verweisen in einer Richtlinienerklärung unter Verwendung eines Amazon-Ressourcennamens (ARN) auf eine Lambda-Ereignisfunktion. Das Format des ARN für eine Funktion hängt davon ab, ob

Sie die gesamte Funktion, eine Funktionsversion (unqualifiziert), eine Funktionsversion oder einen Alias referenzieren.

Bei Lambda-API-Aufrufen können Benutzer eine Version oder einen Alias angeben, indem sie einen Versions-ARN oder Alias-ARN im GetFunction`FunctionName`Parameter übergeben oder einen Wert im GetFunction`Qualifier`Parameter festlegen. Lambda trifft Autorisierungsentscheidungen, indem es das Ressourcenelement in der IAM-Richtlinie sowohl mit dem in API-Aufrufen übergebenen `FunctionNamen` als auch dem `Qualifier` vergleicht. Wenn es keine Übereinstimmung gibt, lehnt Lambda die Anforderung ab.

Unabhängig davon, ob Sie eine Aktion für Ihre Funktion zulassen oder verweigern, müssen Sie die richtigen ARN-Typen für Funktionen in Ihrer Richtlinienanweisung verwenden, um die erwarteten Ergebnisse zu erzielen. Wenn Ihre Richtlinie beispielsweise auf den unqualifizierten ARN verweist, akzeptiert Lambda Anfragen, die auf den unqualifizierten ARN verweisen, lehnt jedoch Anfragen ab, die auf einen qualifizierten ARN verweisen.

> **Note**
>
> Sie können kein Platzhalterzeichen (*) verwenden, um die Konto-ID abzugleichen. Weitere Informationen zur zulässigen Syntax finden Sie unter IAM JSON-Richtlinienreferenz im IAM-Benutzerhandbuch.

Example Zulassen des Aufrufs eines nicht qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction"
        }
    ]
}
```

Wenn Ihre Richtlinie auf einen bestimmten qualifizierten ARN verweist, akzeptiert Lambda Anfragen, die auf diesen ARN verweisen, lehnt aber Anfragen ab, die auf den nicht qualifizierten ARN oder einen anderen qualifizierten ARN verweisen, z. B. `myFunction:2`.

Example Zulassen des Aufrufs eines bestimmten qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

Wenn Ihre Richtlinie mit `:*` einen qualifizierten ARN referenziert, akzeptiert Lambda alle qualifizierten ARN, lehnt jedoch Anfragen ab, die den unqualifizierten ARN referenzieren.

Example Zulassen des Aufrufs eines beliebigen qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

Wenn Ihre Richtlinie mit * einen ARN referenziert, akzeptiert Lambda alle qualifizierten oder unqualifizierten ARNs.

Example Zulassen des Aufrufs eines beliebigen qualifizierten oder nicht qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction*"
        }
    ]
}
```

Im Abschnitt „Ressourcen“ der Richtlinien wird auf dauerhafte Ausführungen verwiesen

Eine dauerhafte Ausführung ist eine Unterressource einer Funktionsversion. Die ARN-Struktur und die politischen Leitlinien für die dauerhafte Ausführung finden Sie unterthe section called “IAM-Berechtigungen für dauerhafte Funktionen”.

Sie können dauerhafte Funktionen nur mit einem qualifizierten Funktionsnamen oder ARN aufrufen. Daher müssen Richtlinien, die den Aufruf dauerhafter Funktionen zulassen, mit qualifizierten ARNs übereinstimmen. Weitere Informationen finden Sie unter the section called “Qualifizierte ARNs erforderlich”.

Wenn Ihre Richtlinie auf die unqualifizierte Funktion ARN verweist, lehnt Lambda Anfragen für dauerhafte Ausführungsaktionen ab. Der unqualifizierte ARN entspricht niemals einem dauerhaften Ausführungs-ARN.

Example Richtlinie, die keinen Zugriff auf dauerhafte Ausführungen gewährt

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": "lambda:GetDurableExecution",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myDurableFunction"
        }
    ]
}
```

Wenn Ihre Richtlinie auf die Funktion ARN mit verweist:*, akzeptiert Lambda Anfragen für dauerhafte Ausführungen jeder Version.

Example Ermöglicht den Zugriff auf dauerhafte Ausführungen jeder Version

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:GetDurableExecution",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myDurableFunction:*"
        }
    ]
}
```

Wenn Ihre Richtlinie auf die Funktion ARN mit verweist:1/durable-execution/*, akzeptiert Lambda nur Anfragen für dauerhafte Ausführungen von Version 1.

Example Ermöglicht den Zugriff auf dauerhafte Ausführungen einer bestimmten Version

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:GetDurableExecution",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myDurableFunction:1/durable-execution/*"
        }
    ]
}
```

> **Aliasmuster stimmen niemals mit ARNs für dauerhafte Ausführungen überein**
>
> Aliasnamen erscheinen niemals in ARNs für dauerhafte Ausführung. Ein Ressourcenmuster wie folgt `my-function:prod/durable-execution/*` nie einer dauerhaften Ausführung, da Lambda den Alias bei der Ausführung der Funktion in eine Versionsnummer auflöst.

Lambda autorisiert die `lambda:ListDurableExecutionsByFunction` Aktion gegen den Funktions-ARN aus der Anfrage, nicht gegen einen dauerhaften Ausführungs-ARN. Die folgenden Regeln gelten für den ARN-Abgleich:

- Eine Richtlinie, die der unqualifizierten Funktion ARN gewährt, entspricht nur Anfragen, die den unqualifizierten Funktionsnamen verwenden.
- Eine Richtlinie, die einen qualifizierten ARN gewährt, entspricht nur Anfragen, die einen qualifizierten Funktionsnamen verwenden.

Example Ermöglicht die Auflistung dauerhafter Ausführungen in beiden Namensformen

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:ListDurableExecutionsByFunction",
            "Resource": [
                "arn:aws:lambda:us-west-2:123456789012:function:myDurableFunction",
                "arn:aws:lambda:us-west-2:123456789012:function:myDurableFunction:*"
            ]
        }
    ]
}
```

Unterstützte IAM-Aktionen und Funktionsverhalten

Aktionen definieren, was durch IAM-Richtlinien zugelassen werden kann. Eine Liste der in Lambda unterstützten Aktionen finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Lambda in der Referenz zur Dienstautorisierung. Wenn eine IAM-Aktion eine Lambda-API-Aktion zulässt, ist der Name der IAM-Aktion in den meisten Fällen derselbe wie der Name der Lambda-API-Aktion, mit den folgenden Ausnahmen:

API-Aktion	IAM-Aktion
Invoke	`lambda:InvokeFunction`
GetLayerVersion GetLayerVersionByArn	`lambda:GetLayerVersion`

Zusätzlich zu den Ressourcen und Bedingungen, die in der Referenz für die Service-Autorisierung definiert sind, unterstützt Lambda die folgenden Ressourcen und Bedingungen für bestimmte Aktionen. Viele davon beziehen sich auf Referenzfunktionen im Ressourcenbereich der Richtlinien. Aktionen, die für eine Funktion verwender werden, können – wie in der folgenden Tabelle beschrieben – nach Funktion, Version oder Alias-ARN auf eine spezifische Funktion eingeschränkt werden.

Action	Ressource	Bedingung
AddPermission RemovePermission Aufrufen (Berechtigung: `lambda:InvokeFunction`)	Funktionsversion Funktionsalias	N/A
ListDurableExecutionsByFunction	Funktionsversion	N/A
UpdateFunctionConfiguration	N/A	`lambda:CodeSigningConfigArn`
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	Funktionsalias	N/A

AWS Lambda-Webfunktionen

Mit AWS Lambda Web Functions können Sie Webanwendungen und API-Backends als serverlose HTTP-Server ausführen. Ihr Code überwacht einen Netzwerkport und verarbeitet Anfragen direkt. Sie können vertraute Node.js Web-Frameworks wieExpress,Hono, oder React Router ohne Lambda-specific Adapter oder Wrapper verwenden. Sie entwickeln und betreiben Ihren Server lokal wie jede andere Node.js Web-App.

Sie stellen Ihren Code als ZIP-Datei bereit. Lambda erstellt einen HTTPS-Endpunkt, verwaltet die Ausführungsumgebungen und skaliert Ihre Anwendung anhand des Datenverkehrs, auch auf Null, wenn sie inaktiv ist. Um die Latenz für global verteilte Benutzer zu reduzieren, können Sie die Lösung für mehrere Benutzer AWS-Regionen hinter einem einzigen Endpunkt bereitstellen. Lambda verwendet latenzbasiertes Routing, um jede Anfrage an eine dieser Regionen zu senden. Sie zahlen für Anfragen, aktive CPU-Zeit, den Speicherverbrauch während der Lebensdauer jeder Ausführungsumgebung und die Datenübertragung.

Themen

- Wann sollten Webfunktionen verwendet werden
- Wie funktionieren Webfunktionen
- Wichtigste Vorteile
- Schlüsselkonzepte
- Eine Lambda-Webfunktion erstellen
- Bereitstellen von Webfunktionen
- HTTP-Anfragen und -Antworten
- Konfiguration von Netzwerken für AWS Lambda Web-Funktionen
- Skalierungsverhalten für Lambda Web Functions
- Lebenszyklus der Ausführungsumgebung
- Laufzeiten der Lambda-Webfunktion
- Konfiguration von Lambda-Webfunktionen
- Berechtigungen für Lambda-Webfunktionen
- Endpunktzugriffskontrolle mit Lambda Web Function
- Verschlüsselung im Ruhezustand für Webfunktionen

- Überwachung von Lambda-Webfunktionen
- Beispielanwendungen für Webfunktionen
- Bewährte Methoden für Lambda-Webfunktionen
- Testen von Lambda-Webfunktionen
- Fehlerbehebung bei Lambda-Webfunktionen

Wann sollten Webfunktionen verwendet werden

Verwenden Sie Webfunktionen für Workloads, die HTTP-Verkehr direkt empfangen und darauf reagieren. Häufige Anwendungsfälle umfassen:

- Webanwendungen — Stellen Sie Full-Stack- oder Frontend-Anwendungen bereit, die HTML-, Asset- und API-Routen von einer einzigen Bereitstellung aus bereitstellen.
- REST- und GraphQL-APIs — Erstellen Sie API-Backends, die hohe Anforderungsvolumina bedienen.
- Microservices — Führen Sie HTTP-Dienste aus, die unabhängig vom Datenverkehr skalieren und auch im Leerlauf auf Null skalieren.
- Server-side Rendern (SSR) — Stellt Seiten bereit, die bei der Anfrage auf dem Server gerendert werden und nicht vorkonfiguriert sind. Eine Webfunktion führt den Produktionsserver Ihres Frameworks direkt aus. Sie verarbeitet SSR-Anfragen, Streaming und API-Routen ohne separate Hosting-Ebene. Es skaliert auch mit dem Verkehr. Dies eignet sich für Anwendungen, die bei jeder Anfrage personalisierte oder sich häufig ändernde Inhalte rendern.
- Streaming-Antworten — Streamen Sie die Ausgabe an den Client, sobald sie erstellt werden, z. B. Antworten aus einem großen Sprachmodell.

Verwenden Sie für ereignisgesteuerte Verarbeitung, z. B. die Verarbeitung von Warteschlangennachrichten oder Datei-Uploads, stattdessen Lambda Event Functions.

> **Note**
>
> AWS Lambda Webfunktionen unterstützen derzeit nur Laufzeiten. Node.js Wenn SieJava,Python, .NET oder eine andere Sprache verwenden, finden Sie the section called “Erstellen Sie Ihre erste Funktion” weitere Informationen unter AWS Lambda Event Functions oder MicroVMs Lambda MicroVMS.

Wie funktionieren Webfunktionen

Eine Webfunktion enthält Revisionen und Endpunkte. Eine Revision ist eine unveränderliche Momentaufnahme Ihres Codes und Ihrer Konfiguration. Ein Endpunkt leitet HTTP-Anfragen an eine oder zwei Revisionen weiter. Lambda führt Ihren Server in Ausführungsumgebungen aus, die jeweils mehrere gleichzeitige Anfragen verarbeiten. Lambda skaliert diese Ausführungsumgebungen mit dem Datenverkehr. Details hierzu finden Sie unter the section called “Schlüsselkonzepte”.

Wichtigste Vorteile

Webfunktionen bieten die folgenden Vorteile für Webanwendungen und API-Backends:

- Standarderfahrung bei der Webentwicklung — Stellen Sie Node.js Standardanwendungen bereit, die auf einem Port lauschen. Sie verwenden vertraute Web-Frameworks und Ihre normale Anwendungsstruktur.
- Adaptive Skalierung — Jede Ausführungsumgebung bedient mehrere gleichzeitige Anfragen, wie ein Standard-Webserver. Infolgedessen verarbeiten weniger Ausführungsumgebungen mehr Verkehr. Bestehende Ausführungsumgebungen können einen geringen Anstieg des Datenverkehrs auffangen, sodass weniger Anfragen auf den Start einer neuen Ausführungsumgebung warten.
- Usage-based Abrechnung — Sie zahlen für Anfragen, die CPU-Zeit, die Ihre Anwendung und die Laufzeit aktiv verbrauchen, den Arbeitsspeicher während der gesamten Lebensdauer der einzelnen Ausführungsumgebungen und die Datenübertragung. Eine Anfrage, die auf eine Datenbankabfrage oder einen externen API-Aufruf wartet, verbraucht während der Wartezeit wenig CPU. Solange die Ausführungsumgebung existiert, nimmt jedoch weiterhin Arbeitsspeicher zu, einschließlich Leerlaufzeiten zwischen Anfragen.
- Antwort-Streaming — Streamen Sie Antworten an Clients, während Ihre Anwendung sie generiert, ohne zusätzliche Konfiguration.
- Auf Null skalieren — Lambda entfernt ungenutzte Ausführungsumgebungen, wenn kein Datenverkehr vorhanden ist, und erstellt neue, wenn Anfragen eintreffen. Eine Anfrage kann warten, bis eine Ausführungsumgebung initialisiert wird. Einzelheiten zur Funktionsweise von Skalierung und Drosselung finden Sie unter. the section called “Skalierung”
- Built-in HTTPS-Endpunkt — Jeder Endpunkt stellt eine HTTPS-Domain mit TLS bereit. Sie wählen für jeden Endpunkt IAM-Authentifizierung oder anwendungsverwaltete Authentifizierung. Details hierzu finden Sie unter the section called “Zugriffskontrolle”. Um eine benutzerdefinierte Domain oder einen benutzerdefinierten AWS WAF Schutz zu verwenden, stellen Sie eine CloudFront Amazon-Distribution vor den Endpunkt.

- Endpoint-level Verkehrskontrollen — Legen Sie ein Limit für die Anforderungsrate (Anfragen pro Sekunde) für jeden HTTPS-Endpunkt fest, das Lambda in jeder Region durchsetzt, sowie eine maximale Anzahl von Ausführungsumgebungen (`maxEnvironments`), die ebenfalls in jeder Region durchgesetzt wird. Zusammen können Sie damit den Traffic begrenzen, den ein Endpunkt akzeptiert. Außerdem können Sie damit einschränken, wie viel Last der Endpunkt an nachgelagerte Abhängigkeiten wie Datenbanken und externe APIs weitergibt. Weitere Informationen finden Sie unter the section called “Skalierung”.

Schlüsselkonzepte

Wenn Sie die folgenden Kernkonzepte verstehen, können Sie Webanwendungen effektiv bereitstellen und verwalten. Auf dieser Seite werden das Ressourcenmodell einer Webfunktion, die Lebenszyklusstatus jeder Ressource und die Art und Weise behandelt, wie Lambda eingehende HTTP-Anfragen behandelt, behandelt.

Das Ressourcenmodell definiert die Struktur einer Webfunktion. Eine Webfunktion ist die Ressource der obersten Ebene. Sie enthält Revisionen (unveränderliche Schnappschüsse Ihres Codes und Ihrer Konfiguration) und Endpunkte (Netzwerkeingangspunkte, die HTTP-Verkehr zu Ihren Versionen weiterleiten).

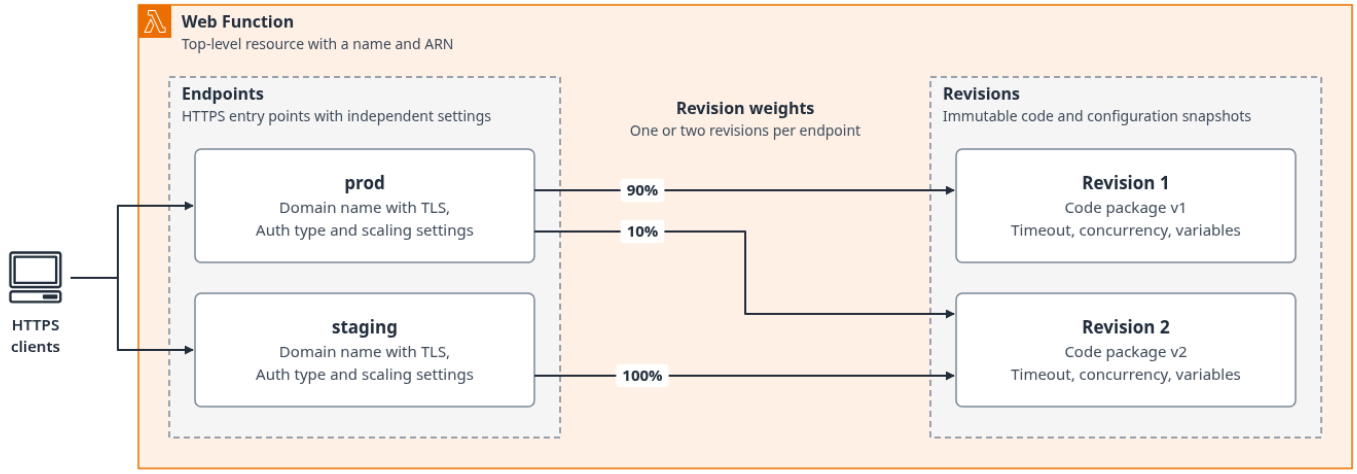
Revisionsstatus und Endpunktstatus verfolgen den Fortschritt dieser Ressourcen, während Lambda sie erstellt, bereitstellt und löscht. Durch die Überwachung dieser Zustände erfahren Sie, wann Lambda die Erstellung, Bereitstellung oder Löschung einer Ressource abgeschlossen hat, und hilft Ihnen bei der Diagnose von Fehlern.

Ausführungsumgebungen sind isolierte Recheninstanzen, auf denen Ihr Webserver und Ihre Anwendung ausgeführt werden. Jede Ausführungsumgebung verarbeitet mehrere gleichzeitige HTTP-Anfragen. Lambda erstellt und entfernt automatisch Umgebungen auf der Grundlage Ihrer Webfunktionskonfiguration und Ihres Datenverkehrs.

Die folgenden Abschnitte enthalten detaillierte Informationen zu den einzelnen Kernkonzepten.

Ressourcenmodell

Eine Lambda-Webfunktion besteht aus drei Ressourcentypen in einer Parent-Child-Hierarchie. Die Webfunktion ist die Ressource der obersten Ebene. Revisionen und Endpunkte sind untergeordnete Ressourcen der Webfunktion. Durch das Bereitstellen von Code oder das Ändern der Revisionskonfiguration wird eine neue Revision erstellt. Das folgende Diagramm zeigt eine Webfunktion, deren Endpunkte den Datenverkehr zu ihren Revisionen weiterleiten.



Web-Funktion

Die Ressource auf oberster Ebene, die Ihre Webanwendung repräsentiert. Eine Webfunktion hat einen Namen, einen ARN und ihren eigenen Lebenszyklusstatus (Ausstehend, Aktiv, Fehlgeschlagen oder Löschen). Sie kann Revisionen und Endpunkte enthalten; alle Revisionen und Endpunkte gehören zu einer einzigen übergeordneten Webfunktion.

Revision

Eine unveränderliche Momentaufnahme Ihres Anwendungscodes und Ihrer Konfiguration. Sie erstellen jedes Mal eine neue Revision, wenn Sie Ihren Code oder Ihre Revisionskonfiguration ändern, z. B. das Anforderungs-Timeout, die maximale Anzahl gleichzeitiger Anfragen pro Ausführungsumgebung oder Umgebungsvariablen. Da Revisionen unveränderlich sind, können Sie jederzeit zu einer früheren Version Ihrer Anwendung zurückkehren, indem Sie den Datenverkehr an eine frühere Version weiterleiten. Bei kanarischen oder blaugrünen Bereitstellungen können Sie den Datenverkehr auf zwei Versionen gleichzeitig verteilen.

Endpunkt

Ein Netzwerkeingangspunkt, der HTTP-Verkehr zu Ihren Revisionen weiterleitet. Jeder Endpunkt hat einen eindeutigen Domainnamen mit TLS, einen Authentifizierungstyp, der den Zugriff steuert, und Skalierungseinstellungen, die die Kapazität definieren. Wenn Sie einen Endpunkt erstellen, wählen Sie seinen Endpunkttyp aus, der seinen geografischen Geltungsbereich festlegt. HomeRegiondient nur dem Verkehr aus der Heimatregion. MultiRegionbedient Datenverkehr aus mehreren Regionen hinter einer globalen Domain, die jede Anfrage auf der Grundlage der Netzwerklatenz nach bestem Wissen und Gewissen an eine Region weiterleitet. PerRegionbedient Traffic aus mehreren Regionen, von denen jede ihre eigene regionale Domain hat, und Sie kontrollieren, wie Kunden jede Region erreichen. Eine Webfunktion kann mehrere

Endpunkte haben, von denen jeder eine unabhängige Konfiguration hat. Sie konfigurieren Revisionsgewichte auf einem Endpunkt, um den Datenverkehr an eine oder zwei aktive Revisionen weiterzuleiten.

Status der Revision

Eine Revision durchläuft während ihres Lebenszyklus die folgenden Zustände. Sie können den aktuellen Status überprüfen, indem Sie die Revisionsdetails in der Lambda-Konsole aufrufen `GetWebFunctionRevision` oder sich diese ansehen.

Status	Description	Wie geht es weiter?
Ausstehend	Lambda erstellt die Revision. Dazu gehört das Herunterladen Ihres Codes von Amazon S3 und die Validierung der Konfiguration.	Die Revision wechselt zu Aktiv, wenn die Erstellung erfolgreich ist, oder zu Fehlgeschlagen, wenn ein Fehler auftritt.
Aktiv	Die Revision wurde erfolgreich erstellt und kann nun hinter einem Endpunkt abgelegt werden.	Die Revision bleibt aktiv, bis Sie sie löschen, Lambda löscht sie während der automatischen Revisions bereinigung (siehethe section called “Revisionsgrenzen und automatische Bereinigung”) oder Sie entfernen den Zugriff auf den Quellcode. Wenn Lambda den Zugriff auf den Code verliert, beispielsweise weil die referenzierte Amazon S3 S3-Objekt version gelöscht wurde, wird die Revision in den Status Fehlgesch lagen verschoben (siehethe section called “Lebenszyklus eines Objekts”).
Fehlgeschlagen	Lambda konnte die Revision nicht erstellen. Probleme, die Lambda beim Absenden der Anfrage feststell t, führen nicht zu einer Fehlgesch lagenen Revision. Wenn der AWS	Einzelheiten finden Sie im `stateReason` Feld in der `GetWebFunctionRevision` Antwort. Erstellen Sie eine neue Version, um das Problem zu

Status	Description	Wie geht es weiter?
	KMS Schlüssel beispielsweise nicht über ausreichende Berechtigungen verfügt, schlägt die Anfrage mit einer Client-Ausnahme fehl und Lambda erstellt keine Revision.	beheben. Fehlgeschlagene Revisionen können den Datenverk ehr nicht bedienen.

Endpunktzustände

Ein Endpunkt hat zwei Statusfelder: Er `state` verfolgt den gesamten Lebenszyklus und `updateStatus` verfolgt Endpunktaktualisierungen nach der ersten Erstellung. Sie können beide überprüfen, indem Sie anrufen`GetWebFunctionEndpoint`.

Status	Description	Wie geht es weiter?
Ausstehend	Lambda erstellt den Endpunkt. Dazu gehören die Zuweisung eines Domainnamens, die Bereitstellung von TLS und die Konfiguration des Routings. Dazu gehört auch die Replikation der Konfiguration auf andere `PerRegion` Endgeräte. `MultiRegion` AWS-Regionen	Der Endpunkt wechselt zu Aktiv, wenn die Erstellung in allen Zielregio nen erfolgreich war, oder zu Fehlgeschlagen, wenn ein Fehler auftritt.
Aktiv	Der Endpunkt ist bereit, HTTP-Verk ehr unter seinem Domainnamen zu empfangen. Clients können Anfragen an den Endpunkt senden.	Der Endpunkt bleibt aktiv. Lambda verfolgt nachfolgende Endpunkta ktualisierungen vor `updateStatus` Ort, ohne den Gesamtstatus des Endpunkts zu ändern.
Fehlgeschlagen	Die Erstellung ist fehlgeschlagen. Bei `MultiRegion` `PerRegion` Endpunkten kann dies auf einen Fehler in einer oder mehreren Zielregionen hinweisen.	Einzelheiten finden Sie im `stateReason` Feld. Überprüfen Sie für `MultiRegion` `PerRegion` Endpunkte die `regionalE ndpoints` Einträge für die `state` und `stateReason` für jede Region.

Status	Description	Wie geht es weiter?
Wird gelöscht	Lambda löscht den Endpunkt und zieht seinen Domainnamen zurück. Für `MultiRegion` `PerRegion` Endpunkte entfernt Lambda den Endpunkt aus allen Regionen.	Nach Abschluss des Löschvorgangs wird der Endpunkt dauerhaft entfernt. Sein Domainname ist zurückgezogen und Lambda weist ihn in keinem Konto einem anderen Endpunkt zu. Dieser Vorgang kann nicht rückgängig gemacht werden.

Für einen vorhandenen Endpunkt bleibt Lambda `Active` während eines Endpunkt-Updates auf `state` eingestellt. Prüfen `updateStatus` Sie, ob das neueste Update `InProgressSuccessful`, oder `Failed` ist. Wenn Sie beispielsweise die Revisionsgewichte eines Endpunkts ändern, `updateStatus` `state` bleibt die `InProgress` Zeit erhaltenActive. Wenn die neuen Gewichtungen in Kraft sind, `updateStatus` ändert sich der Wert von`Successful`.

InProgress

Lambda wendet das Endpunkt-Update an.

Erfolgreich

Das letzte Endpunkt-Update wurde abgeschlossen und ist in Kraft.

Fehlgeschlagen

Das letzte Endpunkt-Update ist fehlgeschlagen. Überprüfen Sie das `updateStatusReason` Feld und finden Sie `regionalEndpoints` weitere Informationen. Für `MultiRegion` und `PerRegion` Endpunkte verwenden einige Regionen möglicherweise die vorherige Konfiguration, während andere Regionen die angeforderte Konfiguration verwenden. Korrigieren Sie die gemeldete Ursache und rufen Sie dann `UpdateWebFunctionEndpoint` erneut mit der vorgesehenen Konfiguration an, um den Endpunkt abzustimmen.

Der Status der Webfunktion, der Revisionsstatus und der Endpunktstatus sind unabhängig. Eine Webfunktion kann sich im Status Aktiv befinden, während einer ihrer Endpunkte noch ausstehend oder ausgefallen ist. In ähnlicher Weise kann eine Revision auch dann aktiv sein, wenn derzeit kein Endpunkt Datenverkehr an sie weiterleitet.

Eine aktive Revision, ein aktiver Endpunkt oder ein erfolgreiches Endpunkt-Update bestätigt nicht, dass Ihr Server gestartet wird. Lambda überprüft nicht, ob Ihr Server zuhört, wenn er einen Endpunkt erstellt oder aktualisiert, um eine Revision zu verwenden. Senden Sie nach jeder Bereitstellung eine Testanfrage und überprüfen Sie Ihre Anwendungsprotokolle.

Ausführungsumgebungen

Eine Ausführungsumgebung ist eine isolierte Recheninstanz, in der Ihr Webserver und Ihre Anwendung ausgeführt werden. Wenn der Datenverkehr an Ihrem Endpunkt ankommt, leitet Lambda Anfragen an Ausführungsumgebungen weiter, in denen der Code der aktiven Revision ausgeführt wird.

Lambda verwaltet Ausführungsumgebungen automatisch:

1. Skalieren — Wenn der Traffic zunimmt, erstellt Lambda zusätzliche Ausführungsumgebungen und startet Ihren HTTP-Server in jeder Umgebung. Lambda skaliert neue Umgebungen vor dem Bedarf, um genügend Spielraum zu haben, sodass Anfragen selten auf den Start einer neuen Umgebung warten (Kaltstart). Wenn eine Webfunktion von Null aus skaliert (die erste Anforderung nach einer Zeit ohne Datenverkehr), kommt es zu einer kurzen Kaltstartverzögerung, während Lambda eine Ausführungsumgebung erstellt. Die Anforderung wartet, bis die Ausführungsumgebung initialisiert wird. Die Anforderung schlägt möglicherweise trotzdem fehl, wenn Ihre Anwendung nicht gestartet wird oder wenn eine Anforderungsrate, eine Ausführungsumgebung oder ein Kontolimit gilt. Details hierzu finden Sie unter the section called “Skalierung”.
2. Herunterskalieren — Wenn der Traffic abnimmt, entfernt Lambda Ausführungsumgebungen, die es nicht mehr benötigt, nachdem die laufenden Anfragen nach bestem Wissen abgeschlossen wurden. Lambda skaliert auf Null-Umgebungen, wenn kein Verkehr vorhanden ist.

Die Anzahl der Ausführungsumgebungen und ihre Parallelität werden durch zwei Einstellungen gesteuert:

- **`maxEnvironments`**(Endpunkteinstellung) — Die maximale Anzahl von Ausführungsumgebungen, die Lambda für den Endpunkt in jeder Region erstellen kann. Diese Einstellung ist optional. Wenn es nicht festgelegt ist, skaliert Lambda die Ausführungsumgebungen nach Bedarf, abhängig von Ihren Kontokontingenten.

- **`maxConcurrencyPerEnvironment`**(Revisionseinstellung) — Die maximale Anzahl von HTTP-Anfragen, die eine einzelne Ausführungsumgebung gleichzeitig verarbeitet. Gültige Werte sind 1—128. Der Standardwert ist 64.

Lambda legt für jede Ausführungsumgebung ein dynamisches Parallelitätslimit fest. Dieses Limit wird niemals überschritten. `maxConcurrencyPerEnvironment` Eine Obergrenze für die Parallelität von Endpunkten wird `maxEnvironments` mit diesem dynamischen Limit multipliziert. Die tatsächliche Kapazität ist möglicherweise niedriger, da Lambda die Parallelität pro Umgebung auf der Grundlage der Ressourcenauslastung anpasst und Endpunkt- und Kontolimits anwendet. Nehmen wir zum Beispiel an, Sie setzen `maxEnvironments` auf 20 und auf 64. `maxConcurrencyPerEnvironment` Der Endpunkt verarbeitet dann bis zu 1.280 gleichzeitige Anfragen in einer Region. Es verarbeitet weniger, wenn Lambda das dynamische Limit unter 64 setzt. Hinweise zu diesen Einstellungen finden Sie unter the section called “Skalierung”.

Bearbeitung von Anfragen

Ihr Web-Funktionscode wird als Standard-HTTP-Server ausgeführt, der auf einem Port lauscht. Lambda sendet eingehende HTTP-Anfragen an Ihren Server und gibt die Antworten an die Clients zurück. Sie können jedes Node.js Web-Framework, jede Middleware oder jeden Router verwenden.

Der Anforderungslebenszyklus funktioniert wie folgt:

1. Ein Client sendet eine HTTP-Anfrage an den Domainnamen des Endpunkts.
2. Lambda authentifiziert die Anfrage (für IamAuthEndpunkte) oder leitet sie weiter (für ApplicationManagedEndpunkte).
3. Lambda leitet die Anfrage auf der Grundlage der Revisionsgewichte des Endpunkts an eine Ausführungsumgebung weiter.
4. Der HTTP-Server der Ausführungsumgebung verarbeitet die Anfrage und gibt eine Antwort zurück.
5. Lambda gibt die Antwort an den Client zurück.

Jede Anfrage kann bis zu 900 Sekunden (15 Minuten) lang ausgeführt werden. Sie können das Timeout mit den `timeoutSeconds` Einstellungen der Revision zwischen 3 und 900 Sekunden konfigurieren. Das Standard-Timeout beträgt 30 Sekunden.

Jede Ausführungsumgebung für Webfunktionen verarbeitet mehrere gleichzeitige Anfragen, sodass Ihr Anwendungscode für die gleichzeitige Verwendung sicher sein muss. Einzelheiten zum Parallelitätsmodell finden Sie unter. the section called “Node.js Ausführungsmodell”

Revisionen von Webfunktionen verwalten

Eine Revision ist eine unveränderliche Momentaufnahme des Codes und der Konfiguration Ihrer Web Function-Anwendung. Um Ihren Code oder eine Revisionseinstellung wie, oder `environmentVariables` zu ändern `timeoutSecondsmaxConcurrencyPerEnvironment`, erstellen Sie eine neue Revision. Ob die neue Version Datenverkehr empfängt, hängt vom automatischen Bereitstellungsmodus der einzelnen Endgeräte ab. Ein Endpunkt im LatestRevisionModus leitet den gesamten Datenverkehr an die neue Version weiter, sobald diese aktiv wird. Ein Endpunkt im deaktivierten Modus behält seine aktuellen Revisionsgewichte bei, bis Sie sie ändern. Da Revisionen unveränderlich sind, können Sie ein Rollback durchführen, indem Sie den Datenverkehr zu einer früheren Version weiterleiten. Die frühere Version muss aktiv bleiben und ihr Code, Schlüssel, Rolle und andere Abhängigkeiten müssen verfügbar bleiben. Hinweise zur Weiterleitung des Datenverkehrs zwischen Versionen finden Sie unterthe section called “Bereitstellungsmuster”.

Eigenschaften der Revision

Jede Revision enthält die folgende Konfiguration:

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
revisionId	Eine eindeutige ID, die Lambda generiert, wenn Sie die Revision erstellen.	–	Nein (generiert)
Revision ARN	Der Amazon-Ressourcenname (ARN) der Revision.	–	Nein (generiert)
Funktion ARN	Der ARN der Webfunktion, zu der die Revision gehört.	–	Nein (generiert)
state	Der Lebenszyklusstatus der Revision (`PendingActive`,, oder`Failed`). Details hierzu finden Sie unter the section called “Status der Revision”.	–	Nein (verwaltet von Lambda)
stateReason	Eine Beschreibung, warum sich die Revision in ihrem aktuellen Zustand	–	Nein (verwalte

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
	befindet, z. B. die Ursache eines Zustands. `Failed`		t von Lambda)
Fehler	Eine Liste von Fehlern, die Lambda beim Erstellen oder Lesen der Revision gefunden hat. Jeder Fehler benennt das betroffene Attribut und enthält einen Fehlercode und eine Meldung. Fehlt, wenn die Revision keine Fehler enthält.	–	Nein (verwaltet von Lambda)
createdAt	Der Zeitstempel, als die Revision erstellt wurde.	–	Nein (generiert)
description	Eine optionale Beschreibung der Revision, die Sie bei der Erstellung angeben.	Keine	Nein (unveränderlich)
BuildConfig	Der Code und die Laufzeit für die Revision. Enthält `codeConfig` die in den folgenden Zeilen beschriebenen `runtimeConfig` Einstellungen.	–	Nein (unveränderlich)
CodeConfig	Der Speicherort des Bereitstellungspakets. Enthält die in den folgenden Zeilen beschriebene `s3Object` Einstellung. Teil von`buildConfig` .	–	Nein (unveränderlich)
S3-Objekt	Der Amazon S3 S3-Speicherort des Bereitstellungspakets. Enthält die `bucket` in den folgenden Zeilen beschriebenen `versionId` Einstellungen, und. `key` Details hierzu finden Sie unter the section called “Kodieren Sie in Amazon S3”. Teil von`codeConfig` .	–	Nein (unveränderlich)

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
bucket	Der Name des Amazon S3 S3-Buckets, der das Bereitstellungspaket enthält. Für den Bucket muss die S3-Versionierung aktiviert sein. Teil von. `s3Object`	–	Nein (unveränderlich)
key	Der Amazon S3 S3-Objektschlüssel des Bereitstellungspakets. Teil von`s3Object`.	–	Nein (unveränderlich)
versionId	Die Versions-ID des Amazon S3 S3-Objekts. Optional. Wenn Sie es weglassen, verwendet Lambda die neueste Version des Objekts zum Zeitpunkt der Erstellung. In jedem Fall bindet Lambda die Revision an eine bestimmte Objektversion. Teil von. `s3Object`	Aktuelle Version	Nein (unveränderlich)
RuntimeConfig	Die Laufzeitkonfiguration. Enthält die in der folgenden Zeile beschriebene `runtime` Einstellung. Teil von`buildConfig` .	–	Nein (unveränderlich)

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
runtime	Die Laufzeit-ID (zum Beispiel`nodejs24.x`). Informationen zu unterstützten Laufzeiten finden Sie unterthe section called “Laufzeiten”. Die Laufzeit-ID für die Revision ist zwar festgelegt, die Laufzeit selbst ist jedoch nicht eingefroren. Lambda wendet automatisch Sicherheitspatches und Bugfixes auf die Runtime an, sodass der Identifier (z. B.`nodejs24.x`) immer auf die neueste gepatchte Version aufgelöst wird. Details hierzu finden Sie unter the section called “Automatisches Runtime-Patching”. Teil von. `runtimeConfig`	–	Nein (unverändert)
ServiceConfig	Die Laufzeiteinstellungen für die Revision. Enthält die `executionRoleArn` in den folgenden Zeilen beschriebenen `telemetryConfig` Einstellungen `maxConcurrencyPerEnvironment` `environmentVariables` ,,, und. `timeoutSeconds`	–	Nein (unverändert)
Hinrichtung RoleArn	Der ARN der IAM-Rolle, den die Web-Funktion zur Laufzeit annimmt. Die Rolle gewährt Berechtigungen für den Zugriff auf AWS Dienste und Ressourcen. Erforderlich Details hierzu finden Sie unter the section called “Rolle „Ausführung“”. Teil von`serviceConfig` .	–	Nein (unverändert)

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
timeoutSeconds	Die maximale Zeit in Sekunden, die eine einzelne HTTP-Anfrage ausgeführt werden kann. Gültige Werte: 3—900. Teil von. `serviceConfig`	30	Nein (unveränderlich)
maxConcurrencyPerEnvironment	Die maximale Anzahl gleichzeitiger HTTP-Anfragen pro Ausführungsumgebung. Gültige Werte: 1—128. Teil von. `serviceConfig`	64	Nein (unveränderlich)
Umgebungsvariablen	Key-value Paare, die Ihrer Anwendung zur Laufzeit zur Verfügung stehen. Wird pro Revision festgelegt und ist nach der Erstellung unveränderlich. Teil von. `serviceConfig`	Keine	Nein (unveränderlich)
TelemetrieConfig	Telemetrieeinstellungen für die Revision. Enthält die in den folgenden Zeilen beschriebene `loggingConfig` Einstellung. Wenn Sie es weglassen `telemetryConfig` , sendet Lambda Ihre Anwendungsprotokolle trotzdem an die Standardprotokollgruppe. Teil von. `serviceConfig`	Keine	Nein (unveränderlich)
Konfiguration protokollieren	Die Protokollierungskonfiguration für die Revision. Enthält die `logGroup` in den folgenden Zeilen beschriebenen `systemLogLevel` Einstellungen, und. `applicationLogLevel` Details hierzu finden Sie unter the section called “Kontrollen für die Protokollierung”. Teil von`telemetryConfig` .	Keine	Nein (unveränderlich)

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
logGroup	Die Amazon CloudWatch Logs-Protokollgruppe, die Ihre Protokolle empfängt. Protokollgruppennamen, die mit beginnen, `aws/` sind reserviert und werden zurückgewiesen. Details hierzu finden Sie unter the section called "Verwenden einer benutzerdefinierten Protokollgruppe". Teil von`loggingConfig` .	`/aws/lambda/web/`*`functi name`*	Nein (unverändert)
AnwendungLogLevel	Der minimale Schweregrad der Protokolle Ihrer Anwendung, die Lambda an Amazon CloudWatch Logs sendet. Gültige Werte: `TRACE`,`DEBUG`,`INFO`, `WARN`,`ERROR`, und`FATAL`. Teil von`loggingConfig` .	`INFO`	Nein (unverändert)
System LogLevel	Der minimale Schweregrad der Lambda-generated Systemprotokolle, die Lambda an Amazon CloudWatch Logs sendet. Gültige Werte: `DEBUG`, `INFO` und `WARN` Teil von. `loggingConfig`	`INFO`	Nein (unverändert)

Property (Eigenschaft)	Description (Beschreibung)	Standard	Veränderbar
km KeyArn	Der ARN des vom Kunden verwalteten Schlüssels, der zum Verschlüsseln von Code und Umgebungsvariablen verwendet wird. Optional. Wenn ein `MultiRegion PerRegion` Oder-Endpunkt den Datenverkehr an die Version weiterleitet, verwenden Sie einen Schlüssel für mehrere Regionen mit einem Replikat in jeder Region des Endpunkts. Details hierzu finden Sie unter the section called “Verschlüsselung im Ruhezustand”.	Keine	Nein (unveränderlich)

Alle Revisionseigenschaften sind nach der Erstellung unveränderlich. Um eine Konfiguration zu ändern, erstellen Sie eine neue Revision mit den aktualisierten Werten.

Eine Revision erstellen

Lambda erstellt eine Revision, wenn Sie aufrufen. `CreateWebFunctionRevision` Sie geben den Speicherort und die Konfiguration des Codes in der Anfrage an. Eine neue Revision kopiert keine Einstellungen aus einer vorhandenen Revision. Geben Sie alle Einstellungen an, die Sie behalten möchten, einschließlich der Umgebungsvariablen. Einstellungen, die Sie auslassen, verwenden ihre Standardwerte. Wenn Sie mit den aktuellen Werten beginnen möchten, führen Sie es aus `get-web-function-revision` und verwenden Sie es `buildConfig` `serviceConfig` mit Ihren Änderungen erneut. Die neue Revision beginnt im Status Ausstehend, während Lambda Ihren Code herunterlädt und daraus ein Codeartefakt vorbereitet. Wenn die Vorbereitung abgeschlossen ist, geht die Revision in den Status Aktiv über. Wenn während der Vorbereitung ein Fehler auftritt, wechselt die Revision in den Status Fehlgeschlagen und kann keinen Datenverkehr verarbeiten. Einzelheiten zu den einzelnen Bundesstaaten finden Sie unterthe section called “Status der Revision”.

Bevor Sie eine Revision erstellen, laden Sie Ihr .zip-Bereitstellungspaket in einen versionierten Amazon S3 S3-Bucket hoch und rufen Sie den ARN einer Ausführungsrolle ab, die Lambda übernehmen kann. Details dazu finden Sie unter the section called “Kodieren Sie in Amazon S3” und the section called “Rolle „Ausführung“”.

Um eine Revision mit aktualisiertem Code zu erstellen (AWS CLI)

1. Führen Sie den folgenden Befehl aus, um eine neue Revision zu erstellen. Ersetzen Sie den S3-Bucket, den Schlüssel, den Rollen-ARN und die Konfigurationswerte durch Ihre eigenen.

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --build-config '{
      "codeConfig": {
        "s3Object": {
          "bucket": "my-bucket",
          "key": "my-app-v2.zip"
        }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role"
    }'
```

   Der Befehl verwendet die folgenden Objekte. Informationen zu den einzelnen Einstellungen und ihrer Standardeinstellung finden Sie unterthe section called “Eigenschaften der Revision”.

   - `buildConfig`enthält `codeConfig` den Amazon S3 S3-Bucket und den Schlüssel Ihres Bereitstellungspakets sowie `runtimeConfig` die Laufzeit (z. B.`nodejs24.x`). Beide sind erforderlich.
   - `serviceConfig`enthält den erforderlichen ARN der Ausführungsrolle`executionRoleArn`, die Ihre Webfunktion bei der Ausführung annimmt. Es kann auch`timeoutSeconds`, `maxConcurrencyPerEnvironment`enthalten`environmentVariables`, und `telemetryConfig`.

   Der Befehl gibt Details zur neuen Version zurück, einschließlich ihrer`revisionId`. Die Revision beginnt im Status Ausstehend.

2. Führen Sie den folgenden Befehl aus, um zu bestätigen, dass die Revision aktiv ist. *my-web-app*Ersetzen Sie es durch Ihren Funktionsnamen und *rev-a1b2c* durch den `revisionId` aus der Antwort.

```
aws lambda-web get-web-function-revision \
    --function-name my-web-app \
```

```
    --revision-id rev-a1b2c
```

Die Revision ist fertig, wann sie `state` ist`Active`. Falls `state` ja`Failed`, `stateReason` beschreibt es die Ursache. Details hierzu finden Sie unter the section called “Status der Revision”.

Direkt nachdem Sie eine Revision erstellt haben, `state` ist dies in der Regel der Fall`Pending`. Um zu warten, bis es soweit ist`Active`, führen Sie es `aws lambda-web wait web-function-revision-active` mit den gleichen Optionen aus.

Um eine Revision mit aktualisierter Konfiguration zu erstellen (AWS CLI)

1. Um die Konfiguration zu ändern, ohne den Code zu ändern, geben Sie dieselbe Codespeicherposition mit den neuen Einstellungen an. Im folgenden Beispiel wird das Timeout auf 60 Sekunden geändert und eine Umgebungsvariable hinzugefügt.

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --build-config '{
      "codeConfig": {
        "s3Object": {
          "bucket": "my-bucket",
          "key": "my-app-v2.zip"
        }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role",
      "timeoutSeconds": 60,
      "maxConcurrencyPerEnvironment": 32,
      "environmentVariables": {
        "DATABASE_URL": "my-database-endpoint",
        "LOG_LEVEL": "info"
      }
    }'
```

2. Führen Sie den folgenden Befehl aus, um zu bestätigen, dass die Revision aktiv ist. *my-web-app*Ersetzen Sie es durch Ihren Funktionsnamen und *rev-a1b2c* durch den `revisionId` aus der Antwort.

```
aws lambda-web get-web-function-revision \
    --function-name my-web-app \
    --revision-id rev-a1b2c
```

Die Revision ist fertig, wann sie `state` ist`Active`. Falls `state` ja`Failed`, `stateReason` beschreibt es die Ursache. Details hierzu finden Sie unter the section called “Status der Revision”.

Direkt nachdem Sie eine Revision erstellt haben, `state` ist dies in der Regel der Fall`Pending`. Um zu warten, bis es soweit ist`Active`, führen Sie es `aws lambda-web wait web-function-revision-active` mit den gleichen Optionen aus.

Nach der Erstellung der Revision hängt es vom automatischen Bereitstellungsmodus des Endpunkts ab, wie der Datenverkehr sie erreicht. Details hierzu finden Sie unter the section called “Bereitstellungsmuster”.

Überarbeitungen auflisten

Verwenden Sie den folgenden Befehl, um alle Revisionen für eine Webfunktion anzuzeigen:

```
aws lambda-web list-web-function-revisions \
    --function-name my-web-app
```

Die Antwort enthält die ID, den Status, den Erstellungszeitstempel und die Konfigurationszusammenfassung jeder Revision. Verwenden Sie die Revisions-ID, wenn Sie die Gewichte der Endpunkt-Revisionen aktualisieren oder eine Revision löschen.

Löschen einer Revision

Sie können eine Revision löschen, die nicht mehr benötigt wird. Wenn Sie eine Revision löschen, wird sie dauerhaft entfernt. Ihr Bereitstellungspaket in Amazon S3 ist nicht betroffen.

> **Important**
>
> Sie können keine Revision löschen, an die ein Endpunkt den Datenverkehr weiterleitet (eine Revision mit einer Gewichtung ungleich Null auf einem beliebigen Endpunkt). Entfernen Sie die Revision aus den Revisionsgewichten aller Endpunkte, bevor Sie sie löschen.

Um eine Revision zu löschen (AWS CLI)

- Führen Sie den folgenden Befehl aus, um eine bestimmte Revision zu löschen.

```
aws lambda-web delete-web-function-revision \
    --function-name my-web-app \
    --revision-id rev-a1b2c
```

Revisionsgrenzen und automatische Bereinigung

Lambda verwendet 50 Revisionen pro Webfunktion als automatisches Bereinigungsziel. Das Ziel ist nicht anpassbar. In der Konsole Service Quotas wird es als Revisions per Web Function (`web-function-revisions-per-function`) angezeigt.

Sie müssen alte Versionen nicht löschen, um innerhalb des Bereinigungsziels zu bleiben. Wenn eine neue Revision aktiv wird und die Gesamtzahl der Revisionen das Bereinigungsziel überschreitet, löscht Lambda automatisch die ältesten Revisionen, die nicht verwendet werden. Eine Revision wird verwendet, wenn ein Endpunkt Datenverkehr an sie weiterleitet, auch wenn sie irgendwo in der `regionalEndpoints` Eigenschaft eines OR-Endpunkts referenziert wird. `MultiRegion PerRegion` Lambda löscht nicht automatisch eine Revision, die gerade verwendet wird, oder die Revision, die gerade aktiv wurde. Die Bereinigung erfolgt asynchron. Die Anzahl der Revisionen kann vorübergehend 50 überschreiten, solange ältere Revisionen noch verwendet werden oder die Bereinigung im Gange ist. Eine Revision, die Sie aus allen Revisionsgewichten entfernen, z. B. ein Rollback-Ziel, kann durch Bereinigung gelöscht werden.

Sie können Revisionen auch jederzeit manuell löschen. Informationen zum Verfahren finden Sie im vorherigen Abschnitt.

Das Bereitstellungspaket jeder Version kann bis zu 512 MB groß sein (entpackt). Da Sie Ihr Bereitstellungspaket in Ihrem eigenen Amazon S3 S3-Bucket bereitstellen und Lambda Ihren Code direkt aus dem Bucket referenziert, gibt es kein separates Code-Speicherkontingent für Web Functions. Details hierzu finden Sie unter the section called “Kodieren Sie in Amazon S3”.

Konfiguration der Revisionseinstellungen

Die gesamte Konfiguration wird pro Revision festgelegt und ist nach der Erstellung unveränderlich. Um eine Einstellung zu ändern, erstellen Sie eine neue Revision mit den aktualisierten Werten. Die folgenden Einstellungen steuern das Ausführungsverhalten.

Informationen zu den Einstellungen für das Anforderungstimeout und die Umgebungsvariablen finden Sie unter the section called “Zeitüberschreitung” und the section called “Umgebungsvariablen” inthe section called “Konfiguration von Webfunktionen”.

Parallelität pro Umgebung

Die `maxConcurrencyPerEnvironment` Einstellung steuert, wie viele HTTP-Anfragen eine einzelne Ausführungsumgebung gleichzeitig verarbeitet. Gültige Werte sind 1—128. Der Standardwert ist 64.

Bei I/O-bound Anwendungen (viele Anfragen warten auf Datenbank- oder Netzwerkaufrufe) maximiert ein höherer Wert die Auslastung. Bei CPU-bound Anwendungen reduziert ein niedrigerer Wert die Anzahl von Konflikten. Weitere Anleitungen finden Sie unter the section called “Skalierung”.

Endpunkte für Lambda-Webfunktionen

Ein Endpunkt ist eine untergeordnete Ressource einer Webfunktion, die einen Netzwerkeingangspunkt für HTTP-Verkehr bereitstellt. Jeder Endpunkt enthält einen eindeutigen Domainnamen mit TLS und einen Authentifizierungstyp. Es verfügt auch über Einstellungen für geografische Reichweite, Skalierung, Drosselung und Datenverkehrs-Routing. Eine Webfunktion kann bis zu 10 Endpunkte mit jeweils unabhängiger Konfiguration haben.

In den folgenden Abschnitten werden Endpunkttypen, Domainnamen, Authentifizierung und Replikation erläutert. Informationen zu Einstellungen für Skalierung und Drosselung finden Sie unter. the section called “Skalierung” Informationen zu den Modi für die automatische Bereitstellung und zum Routing des Datenverkehrs finden Sie unterthe section called “Bereitstellungsmuster”.

Endpunkttypen

Der Endpunkttyp bestimmt, wie Lambda Ihre Webfunktion bereitstellt AWS-Regionen und wie der Datenverkehr dorthin geleitet wird. Sie wählen den Endpunkttyp aus, wenn Sie den Endpunkt erstellen. Der Endpunkttyp kann nach der Erstellung nicht geändert werden.

HomeRegion

> Führt Ihre Webfunktion in einer einzigen aus AWS-Region. Lambda bedient Anfragen aus der Region, in der Sie den Endpunkt erstellt haben. Lambda weist dem Endpunkt einen einzelnen regionalen Domainnamen zu.
>
> Verwenden Sie dies HomeRegionfür Anwendungen, für die keine Verfügbarkeit in mehreren Regionen erforderlich ist, oder wenn Sie mit der einfachsten Konfiguration beginnen möchten.

MultiRegion

Führt Ihre Webfunktion in mehreren AWS-Regionen Schritten aus. Sie geben die Zielregionen an, wenn Sie den Endpunkt erstellen. Lambda fügt die Heimatregion automatisch hinzu. Der Endpunkt muss mindestens eine weitere Region enthalten. Um den Verkehr an eine Revision weiterzuleiten, legen Sie die Revisionsgewichte des Endpunkts fest. Lambda repliziert dann Ihren Code und Ihre Konfiguration automatisch in alle angegebenen Regionen. Lambda weist einen einzigen globalen Domainnamen zu. Es verwendet latenzbasiertes Routing, um jede Anfrage an die Region zu senden, die normalerweise die niedrigste Latenz für den Anrufer bietet. Lambda führt kein auf Integrität basierendes Failover durch.

Sie verwalten die Webfunktion von einer einzigen Heimatregion aus. Lambda kümmert sich um die Replikation in die anderen Regionen. Clients stellen eine Verbindung zu einem Domainnamen her.

Wird MultiRegionfür globale Anwendungen verwendet, die eine geringe Latenz für Benutzer an mehreren geografischen Standorten benötigen.

PerRegion

Führt Ihre Webfunktion in mehreren aus AWS-Regionen, mit einem separaten regionalen Domainnamen in jeder Region. Sie geben die Zielregionen an, wenn Sie den Endpunkt erstellen. Lambda fügt die Heimatregion automatisch hinzu. Der Endpunkt muss mindestens eine weitere Region enthalten. Lambda richtet den Endpunkt in jeder angegebenen Region automatisch ein. Jede Region hat ihren eigenen unabhängigen Domainnamen.

Im MultiRegionGegensatz dazu führt Lambda kein automatisches latenzbasiertes Routing durch. Sie verwalten, wie der Verkehr die einzelnen regionalen Domänen erreicht, z. B. mit Ihrer eigenen Route-53-Routing-Richtlinie oder einer Verteilung. CloudFront

Verwenden Sie diese Option, PerRegionwenn Sie die Kontrolle über die Routing-Entscheidungen pro Region benötigen, z. B. um geografische Einschränkungen, benutzerdefinierte Failover-Logik oder Region-specific Verkehrsrichtlinien zu implementieren.

Die Regionen und der Endpunkttyp können nach der Erstellung nicht aktualisiert werden. Um diese zu ändern, erstellen Sie einen neuen Endpunkt mit der gewünschten Konfiguration, migrieren Sie den Datenverkehr auf den Domainnamen des neuen Endpunkts und löschen Sie dann den alten Endpunkt.

Einzelheiten darüber, wie Lambda Ihre Webfunktion regionsübergreifend repliziert und den Verkehr weiterleitet, finden Sie unter. the section called “Multi-Region Einsatz”

Domainname

Lambda weist jedem Endpunkt einen eindeutigen Domainnamen zu, wenn Sie ihn erstellen. Clients senden HTTPS-Anfragen an diese Domain, um Ihre Webfunktion aufzurufen. Die Domain unterstützt HTTPS und Lambda verwaltet das TLS-Zertifikat für die Domain. Sie erstellen, erneuern oder verwalten Zertifikate nicht selbst.

Das Format des Domainnamens hängt vom Endpunkttyp ab:

- HomeRegionund PerRegion— Jede Region hat eine Domain im folgenden Format:

  ```
  domain-id.shard-id.lambda-web.region.on.aws
  ```

  Beispiel: `bcdfghjklmnpqrst.b7t.lambda-web.ap-northeast-3.on.aws`

- MultiRegion— Eine einzige globale Domain im Format:

  ```
  domain-id.lambda-web.global.on.aws
  ```

  Beispiel: `bcdfghjklmn.lambda-web.global.on.aws`

  Lambda verwendet latenzbasiertes Routing, um für jeden Client eine Region auszuwählen.

Sie können den Domainnamen aus der `GetWebFunctionEndpoint` Antwort oder von der Lambda-Konsole abrufen, nachdem der Endpunkt den Status Aktiv erreicht hat.

> **Note**
>
> Webfunktions-Endpunkte unterstützen das direkte Anhängen eines benutzerdefinierten Domainnamens nicht. Clients, die die Domäne des Endpunkts aufrufen, müssen den Lambda-allocated Domainnamen verwenden. Um Traffic auf Ihrer eigenen Domain bereitzustellen, stellen Sie eine CloudFront Amazon-Distribution vor den Endpunkt und verwenden Sie den Lambda-allocated Domainnamen als Ursprung.

Authentifizierungstyp

Die `authType` Einstellung steuert, ob Lambda eingehende HTTP-Anfragen authentifiziert, bevor sie an Ihre Anwendung weitergeleitet werden. Sie können diese Einstellung auf einem vorhandenen Endpunkt aktualisieren.

ApplicationManaged

Lambda authentifiziert eingehende Anfragen nicht. Alle HTTP-Anfragen werden direkt an Ihre Anwendung weitergeleitet. Sie sind für die Authentifizierung in Ihrem Anwendungscode verantwortlich.

IamAuth

Lambda validiert die Signatur jeder Anfrage. Signieren Sie Anfragen mit AWS Signature Version 4 (Sigv4) für PerRegionDomains HomeRegionund mit Signature Version 4a (SigV4a) für eine globale Domain. MultiRegion Nur Anrufer mit der entsprechenden `lambda:InvokeWebFunctionEndpoint` Berechtigung können den Endpunkt aufrufen.

Vollständige Informationen zur Authentifizierung, einschließlich CloudFront Integrations- und ressourcenbasierter Richtlinien, finden Sie unter. the section called “Zugriffskontrolle”

Auto-deployment Modus

Die `autoDeploymentMode` Einstellung steuert, ob Lambda bei jeder Bereitstellung Ihrer Anwendung den Datenverkehr automatisch an neue Versionen weiterleitet. Sie können diese Einstellung auf einem vorhandenen Endpunkt aktualisieren.

Wenn Sie `autoDeploymentMode` in einer API-Anfrage weglassen, wird standardmäßig auf. `Disabled` Die Lambda-Konsole und die `aws lambda-web deploy` Befehlsverwendung `LatestRevision` für neue `HomeRegion` Endpunkte. `LatestRevision`wird nur für `HomeRegion` Endgeräte unterstützt, weshalb Endpunkte immer `MultiRegion` verwendet `PerRegion` werden. `Disabled` Geben Sie bei Verwendung `Disabled` `revisionWeights` die Gesamtzahl von 100 an.

LatestRevision

Lambda leitet automatisch 100% des Datenverkehrs an die neueste aktive Version weiter. Wird für Entwicklungs- und Staging-Umgebungen verwendet. LatestRevision Der Modus ist nur für HomeRegionEndpoints verfügbar.

Deaktiviert

Lambda ändert das Traffic-Routing nicht, wenn Sie eine neue Revision erstellen. Sie steuern das Routing manuell mit Revisionsgewichten. Wird für Produktionsumgebungen verwendet. Der deaktivierte Modus ist für alle Endpunkttypen verfügbar. MultiRegionund PerRegionEndgeräte verwenden immer den deaktivierten Modus.

Einzelheiten zu Bereitstellungsmustern, Rollback und Moduswechselverhalten finden Sie unter. the section called “Bereitstellungsmuster”

Skalierung und Drosselung

Jeder Endpunkt verfügt über Skalierungs- und Drosselungseinstellungen, die die Kapazität steuern. Sie können diese Einstellungen auf einem vorhandenen Endpunkt mit aktualisieren. `update-web-function-endpoint` Lassen Sie den aktuellen Wert `--throttle-config` weg `--scaling-config` oder behalten Sie ihn bei, oder übergeben Sie ein leeres Objekt (`'{}'`), um einen zuvor festgelegten Wert zu löschen. Um eine Ratenbegrenzung aufzuheben, übergeben Sie ein leeres `--throttle-config` Objekt. Ein `rateLimit` Wert von 0 blockiert den gesamten Verkehr.

- **`maxEnvironments`** – Optional. Die maximale Anzahl von Ausführungsumgebungen für den Endpunkt in jeder Region. Wenn nicht festgelegt, wird der Endpunkt nach Bedarf innerhalb der Kontokontingente skaliert.
- **`rateLimit`**— Maximale Anzahl von Anfragen pro Sekunde. Diese Einstellung ist optional. Wenn es nicht festgelegt ist, wird der Endpunkt durch das Kontingent für die Anforderungsrate pro Endpunkt von 10.000 RPS und durch die Kontokontingente begrenzt. Das Ratenlimit gilt für den gesamten Endpunkt und für alle Ausführungsumgebungen. Es handelt sich nicht um ein Limit pro Umgebung. Für MultiRegionPerRegionEndgeräte setzt jede Region das Limit unabhängig voneinander durch. Überschüssige Anfragen erhalten HTTP 429.

Einzelheiten dazu, wie diese Einstellungen mit der Parallelität auf Revisionsebene interagieren, sowie Anleitungen zur Auswahl von Werten finden Sie unter. the section called “Skalierung”

Endgeräte erstellen und verwalten

Erstellen eines Endpunkts

Um einen HomeRegion Endpunkt zu erstellen (AWS CLI)

- ```
  aws lambda-web create-web-function-endpoint \
      --function-name my-web-app \
      --endpoint-name my-endpoint \
      --endpoint-type HomeRegion \
      --auth-type ApplicationManaged \
      --auto-deployment-mode LatestRevision
  ```

Um einen MultiRegion Endpunkt zu erstellen (AWS CLI)

- ```
  aws lambda-web create-web-function-endpoint \
      --function-name my-web-app \
      --endpoint-name my-global-endpoint \
      --endpoint-type MultiRegion \
      --regions us-east-1 eu-west-1 ap-northeast-1 \
      --auth-type ApplicationManaged \
      --revision-weights '[{"revisionId": "rev-a1b2c", "weight": 100}]'
  ```

  MultiRegion und PerRegion Endpunkte erfordern Revisionsgewichte, da die LatestRevisionautomatische Bereitstellung für sie nicht verfügbar ist. Weitere Informationen finden Sie unter the section called “Bereitstellungsmuster”.

Auflisten von Endpunkten

```
aws lambda-web list-web-function-endpoints \
    --function-name my-web-app
```

Die Antwort umfasst den Namen, den Typ, den Domänennamen, den Status und die Konfiguration jedes Endpunkts.

Endpunktdetails abrufen

```
aws lambda-web get-web-function-endpoint \
    --function-name my-web-app \
```

```
    --endpoint-name my-endpoint
```

Die Antwort umfasst den Domainnamen, den Status, den Authentifizierungstyp, den automatischen Bereitstellungsmodus, die Skalierungseinstellungen, die Revisionsgewichte `MultiRegion` und (für `PerRegion` Endpunkte) den Status und den Aktualisierungsstatus jeder Region im `regionalEndpoints` Feld.

Aktualisierung eines Endpunkts

Sie können die folgenden Eigenschaften auf einem vorhandenen Endpunkt aktualisieren:

- `description`
- `authType`
- `autoDeploymentMode`
- `revisionWeights`
- `scalingConfig`
- `throttleConfig`

Sie können die folgenden Eigenschaften nicht aktualisieren. Um diese zu ändern, erstellen Sie einen neuen Endpunkt mit der gewünschten Konfiguration, migrieren Sie den Datenverkehr auf den Domainnamen des neuen Endpunkts und löschen Sie dann den alten Endpunkt.

- `endpointType` (HomeRegion, MultiRegion, PerRegion)
- `regions`(die Zielregionen des Endpunkts)

Löschen eines Endpunkts

```
aws lambda-web delete-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

Wenn Sie einen Endpunkt löschen, wechselt er in den Status Löschen. Lambda beendet die Weiterleitung neuer Anfragen an den Endpunkt und zieht seinen Domainnamen zurück. Für einen `PerRegion` Endpunkt stellt Lambda den Domainnamen in jeder Region zurück. Lambda weist die Domainnamen eines gelöschten Endpunkts in keinem Konto einem anderen Endpunkt zu. Lambda leitet Anfragen während des Fluges ab. Es ermöglicht die Fertigstellung von Anfragen, die

bereits in Bearbeitung sind, und lehnt neue Anfragen an die Domainnamen des Endpunkts ab. Für `MultiRegion` `PerRegion` Endpunkte entfernt Lambda den Endpunkt aus allen Regionen. Nach Abschluss des Löschvorgangs entfernt Lambda den Endpunkt dauerhaft.

Durch das Löschen eines Endpunkts werden weder die Webfunktion noch ihre Revisionen gelöscht. Revisionen stehen weiterhin für die Verwendung durch andere Endpunkte zur Verfügung.

Eine Lambda-Webfunktion erstellen

Um eine App mit Lambda Web Functions bereitzustellen, erstellen Sie eine Webfunktion mit einer Revision und einem Endpunkt für den Empfang von Traffic. Die Konsole und der `aws lambda-web deploy` Befehl können die Webfunktion, ihre erste Revision und ihren ersten Endpunkt erstellen. Revisionen sind unveränderliche Schnappschüsse Ihres Codes und Ihrer Konfiguration. Ein Endpunkt leitet Anfragen an eine aktive Revision weiter. Weitere Informationen zu diesen Ressourcen finden Sie unter the section called “Schlüsselkonzepte”. In diesem Thema wird gezeigt, wie Sie sie mithilfe der Lambda-Konsole und der AWS CLI erstellen.

Voraussetzungen

Stellen Sie vor dem Beginn sicher, dass Sie über das Folgende verfügen:

- Ein AWS Konto. Informationen zur Erstellung eines Kontos finden Sie im Referenzhandbuch zur AWS Kontoverwaltung unter Vergleich der Anmeldeoptionen.
- Berechtigungen zur Verwaltung von Webfunktionen, zur Übergabe der Ausführungsrolle an Lambda (`iam:PassRole`) und zur Erstellung oder Verwendung der unterstützenden Amazon S3- und IAM-Ressourcen.

Die Konsole erstellt oder verwendet die Ressourcen, die Ihre Web Function benötigt, einschließlich einer Ausführungsrolle und eines Amazon S3 S3-Buckets für Ihren Code.

Um dem AWS CLI Verfahren zu folgen, benötigen Sie außerdem:

- AWS CLI Version 2, in einer Version, die die `aws lambda-web` Befehle enthält. Um dies zu überprüfen, führen Sie es aus`aws lambda-web help`. Wenn Sie das AWS CLI mit einem offiziellen Installationsprogramm oder Installationsskript installiert haben, führen Sie es aus, `aws update` um auf die neueste Version zu aktualisieren. Andernfalls finden Sie weitere Informationen unter Installation oder Aktualisierung auf die neueste Version von AWS CLI.

- AWS Anmeldeinformationen und eine für die AWS-Region konfigurierte AWS CLI.
- Für die `aws lambda-web deploy` Berechtigung zum Aufrufen`sts:GetCallerIdentity`, für `s3:ListBucket` den Deployment-Bucket und `s3:PutObjects3:GetObject`,, und `s3:GetObjectVersion` für seine Objekte.
- Es `aws lambda-web deploy` erstellt die Standardressourcen, die `iam:GetRole` `iam:CreateRole` Aufrufrechte`iam:PutRolePolicy`,`s3:CreateBucket`,,`s3:PutBucketPolicy`, und`s3:PutBucketVersioning`.
- (Optional) Node.js 24 oder später für die lokale Entwicklung. Informationen zur Installation Node.js finden Sie Node.js auf der Node.js Website unter Herunterladen.

Eine Webfunktion (Konsole) erstellen

Verwenden Sie die Lambda-Konsole, um eine Webfunktion, ihre erste Version und ihren ersten Endpunkt für den Empfang von Traffic zu erstellen.

Um eine Webfunktion und einen Endpunkt (Konsole) zu erstellen

1. Öffnen Sie die Lambda-Konsole unter https://console.aws.amazon.com/lambda/.
2. Wählen Sie im Navigationsbereich Web Functions aus.
3. Wählen Sie Webfunktion erstellen.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie unter Funktionsname einen Namen für Ihre Webfunktion ein.
   b. Behalten Sie für Runtime die Standard-Runtime bei.
   c. Behalten Sie für die Codequelle die Vorlage Hello World bei, um mit einer vorkonfigurierten Beispielanwendung zu beginnen. Um stattdessen Ihren eigenen Code bereitzustellen, wählen Sie ZIP-Datei hochladen oder Amazon S3.
5. Gehen Sie unter Endpunktkonfiguration wie folgt vor:
   a. Geben Sie unter Endpunktname einen Namen für den Endpunkt ein.
   b. Behalten Sie für Typ den Wert bei Home-Region. Ein Home-Region Endpunkt wird in einer einzigen Region ausgeführt, was dies zur einfachsten Konfiguration für Ihre erste Web-Funktion macht. Informationen zu Optionen für mehrere Regionen finden Sie unterthe section called “Endpunkte”.

> **Note**
>
> Sie können den Endpunkttyp nach der Erstellung nicht ändern.

c. Wählen Sie als Authentifizierungstyp die Option Application-managed Authentifizierung aus. Mit dieser Option leitet Lambda alle Anfragen an Ihre App weiter, sodass Ihre App die Authentifizierung übernimmt. Um stattdessen den Zugriff einzuschränken, wählen Sie die IAM-Authentifizierung, bei der Anrufer Anfragen signieren müssen. Anfragen an `HomeRegion` und `PerRegion` Endpunkte verwenden AWS Signature Version 4 (Sigv4). Anfragen an `MultiRegion` Endpunkte verwenden Signature Version 4a (SigV4a). Weitere Informationen finden Sie unter the section called “Zugriffskontrolle”.

d. Behalten Sie für die automatische Bereitstellung die neueste Version bei. In diesem Modus leitet Lambda den gesamten Datenverkehr automatisch an jede neue Revision weiter, sobald sie aktiv wird. Wählen Sie für die manuelle Datenverkehrskontrolle mit Revisionsgewichten die Option Deaktiviert. Weitere Informationen finden Sie unter the section called “Bereitstellungsmuster”.

6. (Optional) Erweitern Sie Zusätzliche Konfigurationen, um die Standardeinstellungen zu überschreiben. Standardmäßig erstellt Lambda eine Ausführungsrolle mit der Berechtigung zum Hochladen von Protokollen auf Amazon und einen Amazon S3 S3-Bucket in Ihrem Konto CloudWatch, um Ihren Code zu speichern. In diesem Abschnitt können Sie stattdessen eine bestehende IAM-Rolle oder Ihren eigenen Amazon S3 S3-Bucket auswählen. Außerdem können Sie die Verschlüsselung mit einem vom Kunden verwalteten Schlüssel, der maximalen Parallelität pro Umgebung, Umgebungsvariablen und Tags konfigurieren.
7. Wählen Sie Webfunktion erstellen.

Wenn Sie Create Web Function wählen, erstellt Lambda die Web Function-Ressource zusammen mit ihrer ersten Revision und gleichzeitig den Endpunkt. Die Revision beginnt im Status Ausstehend, während Lambda Ihren Code herunterlädt und daraus ein Codeartefakt vorbereitet und dann in den Status Aktiv übergeht. Der Endpunkt startet außerdem im Status Ausstehend. Die Web-Funktion, ihre Revision und ihr Endpunkt durchlaufen diese Zustände jeweils unabhängig voneinander. Weitere Informationen zu Revisionen und Lebenszyklusstatus finden Sie unterthe section called “Schlüsselkonzepte”.

Der Endpunkt beginnt mit der Bereitstellung von Datenverkehr, wenn er den Status Aktiv erreicht und es eine aktive Version gibt, an die er weitergeleitet werden kann. Um den Domainnamen des

Endpunkts zu finden, wählen Sie auf der Seite mit den Funktionsdetails die Registerkarte Endpoints und dann den Endpoint aus. Um die App auf Ihrer eigenen Domain bereitzustellen, stellen Sie eine CloudFront Amazon-Distribution vor den Endpunkt. Weitere Informationen zu Endpunkttypen und Domainnamen finden Sie unterthe section called “Endpunkte”.

Um zu bestätigen, dass Ihre App Traffic bereitstellt, senden Sie eine Anfrage an den Domainnamen. Da Sie sich für die Application-managed Authentifizierung entschieden haben, können Sie eine einfache Anfrage senden, ohne sie zu signieren.

```
curl https://your-endpoint-domain-name/
```

Eine erfolgreiche Antwort bedeutet, dass Ihre Webfunktion Traffic empfängt und Ihre App Anfragen bearbeitet.

Eine Webfunktion erstellen (AWS CLI)

Erstellen und implementieren Sie eine Webfunktion von Ihrem Terminal aus. Der Startbefehl generiert einen kleinen Node.js HTTP-Server, stellt ihn bereit und gibt eine HTTPS-Endpunkt-URL zurück.

Stellen Sie Ihre Webfunktion bereit und testen Sie sie

Um eine Starter-Webfunktion bereitzustellen und zu testen

1. Führen Sie den folgenden Befehl aus:

   ```
   aws lambda-web deploy --name my-web-app --hello-world
   ```

   `--hello-world`schreibt die Startdateien in `./my-web-app` und erstellt das Verzeichnis, falls es nicht existiert. Der Starter enthält`server.ts`, einen TypeScript HTTP-Server und`package.json`,`README.md`, und`AGENTS.md`.
2. Wenn die Bereitstellung abgeschlossen ist, kopieren Sie die Endpunkt-URL aus der Ausgabe. Die URL sieht etwa wie folgt aus:

   ```
   https://bcdfghjklmnpqrst.b7t.lambda-web.us-east-1.on.aws
   ```
3. Senden Sie eine Anfrage an die Endpunkt-URL, die Sie im vorherigen Schritt kopiert haben. *`endpoint-url`*Durch diese URL ersetzen:

   ```
   curl endpoint-url
   ```

Der Starter gibt eine Antwort zurück, die der folgenden ähnelt:

```
{"message":"Hello from Lambda Web!","path":"/"}
```

Standardmäßig akzeptiert der Starter-Endpunkt Anfragen ohne Authentifizierung. Um eine IAM-Autorisierung zu verlangen, fügen Sie hinzu`--auth-type IamAuth`. Informationen zur Steuerung des Zugriffs auf Ihren Endpunkt finden Sie unterthe section called “Zugriffskontrolle”.

Was der Befehl erzeugt

Bei der ersten Bereitstellung lautet der Befehl:

- Erstellt einen Amazon S3 S3-Bucket mit aktivierter Versionierung für Bereitstellungspakete oder verwendet ihn erneut.
- Erstellt eine IAM-Ausführungsrolle mit Amazon CloudWatch Logs-Berechtigungen oder verwendet sie erneut.
- Erstellt die Web-Funktion und ihre erste Revision.
- Erzeugt einen `HomeRegion` Endpunkt mit dem Namen `dev` und wartet darauf, dass er wird`Active`. Um einen anderen Namen zu verwenden, fügen Sie hinzu`--endpoint-name`.

Eine Webfunktion enthält unveränderliche Versionen Ihres Codes und Ihrer Konfiguration. Bei späteren Bereitstellungen erstellt der Befehl eine weitere Revision. Der `dev` Endpunkt verwendet den `LatestRevision` Modus und stellt die Revision bereit, nachdem er aktiv geworden ist.

Stellen Sie eine Änderung bereit

Nachdem Sie die Starter-Anwendung bearbeitet haben, führen Sie Folgendes aus:

```
aws lambda-web deploy \
  --name my-web-app \
  --code ./my-web-app
```

Wenn Sie diesen Befehl ausführen, wird Ihr aktualisiertes Paket hochgeladen, eine Revision erstellt und es wird darauf gewartet, dass die neue Version erscheint. `Active` Der `dev` Endpunkt ist auf Revision `autoDeploymentMode` eingestellt `LatestRevision` und stellt die Revision automatisch bereit. Die URL bleibt dieselbe.

Um Ihre eigene App anstelle des Starters bereitzustellen, übergeben Sie das zugehörige Verzeichnis an`--code`. Ihr Server muss den Host und den Port in der `AWS_LAMBDA_HTTP_ENDPOINT` Umgebungsvariablen abhören. Standardmäßig wird die Runtime `index.js` im Stammverzeichnis des Bereitstellungspakets geladen. Um eine andere Eintragsdatei zu verwenden, legen Sie die `AWS_LAMBDA_NODEJS_ENTRYPOINT` Umgebungsvariable fest. Lambda überprüft nicht, ob Ihr Server bei der Erstellung der Revision zuhört. Daher können die Revision und der Endpunkt auch dann angezeigt werden, `Active` wenn Anfragen fehlschlagen. Weitere Informationen finden Sie unter the section called “Laufzeiten”.

Bestehende verwenden AWS Ressourcen

Wenn Sie bereits einen Amazon S3 S3-Bucket oder eine IAM-Ausführungsrolle verwalten, verwenden Sie den folgenden Befehl, um diese Ressourcen wiederzuverwenden, anstatt Standardwerte zu erstellen. Für den Bucket muss die Versionierung aktiviert sein und eine Bucket-Richtlinie, die dem Lambda-Service Principal (`lambda.amazonaws.com`) `s3:GetObject` und gewährt. `s3:GetObjectVersion` Der Befehl ändert nicht die Konfiguration eines von Ihnen angegebenen Buckets. Informationen zu den Bucket-Anforderungen finden Sie unterthe section called “Bucket-Richtlinie für Lambda”. Ersetzen Sie die Beispiel-Ressourcenwerte durch Ihre eigenen.

```
aws lambda-web deploy \
  --name my-web-app \
  --code ./my-web-app \
  --bucket-name amzn-s3-demo-bucket \
  --execution-role-arn arn:aws:iam::111122223333:role/my-web-app-execution-role
```

Die Ausführungsrolle muss es Lambda ermöglichen, sie zu übernehmen, und sie muss die Berechtigungen gewähren, die Ihre Anwendung benötigt. Weitere Informationen finden Sie unter the section called “Rolle „Ausführung“”.

Sie können beide Ressourcenoptionen weglassen, um Standardwerte `deploy` erstellen zu lassen. Wenn eine Webfunktion ohne `--hello-world` Endpunkt `deploy` erstellt oder ein Endpunkt hinzugefügt wird, werden Sie aufgefordert, zur Bestätigung die Eingabetaste zu drücken. Fügen Sie in einem Skript oder einer CI/CD Pipeline ohne Aufforderung `--create` zur Bestätigung hinzu.

Verwenden Sie die API-Befehle direkt

Der `deploy` Befehl lädt Ihren Code auf Amazon S3 hoch und erstellt die Webfunktion, die Revision und den Endpunkt für Sie. Um jeden Schritt selbst zu steuern, verwenden Sie die AWS CLI Befehle, die den API-Operationen zugeordnet sind, wie `create-web-functioncreate-web-function-`

`revision`, und`update-web-function-endpoint`. Ersetzen Sie in den folgenden Befehlen jeden Platzhalter für Benutzereingaben durch Ihre eigenen Informationen.

Um eine Webfunktion mit den API-Befehlen zu erstellen (AWS CLI)

1. Erstellen Sie ein ZIP-Bereitstellungspaket aus Ihrem App-Verzeichnis, wobei sich Ihre Eintragsdatei im Stammverzeichnis des Pakets befindet. Laden Sie das Paket in einen Amazon S3 S3-Bucket hoch, der die Anforderungen in erfülltthe section called “Voraussetzungen”. Die folgenden Befehle erstellen das Paket und verwenden es AWS CLI , um es hochzuladen.

   ```
   cd my-app
   zip -r ../my-app.zip .
   aws s3 cp ../my-app.zip s3://amzn-s3-demo-bucket/my-app.zip
   ```

2. Verwenden Sie den folgenden AWS CLI Befehl, um die Web-Funktion mit ihrer ersten Revision und ihrem ersten Endpunkt zu erstellen. In`revisionConfig`, `buildConfig` gibt den Speicherort und die Laufzeit des Codes `serviceConfig` an und gibt die Ausführungsrolle an. Die `endpointConfig` Eigenschaft definiert den Endpunkt.

   ```
   aws lambda-web create-web-function \
       --function-name my-web-app \
       --revision-config '{
         "buildConfig": {
           "codeConfig": {
             "s3Object": { "bucket": "amzn-s3-demo-bucket", "key": "my-app.zip" }
           },
           "runtimeConfig": { "runtime": "nodejs24.x" }
         },
         "serviceConfig": {
           "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-app-execution-
   role"
         }
       }' \
       --endpoint-config '{
         "endpointName": "my-endpoint",
         "endpointType": "HomeRegion",
         "authType": "ApplicationManaged",
         "autoDeploymentMode": "LatestRevision"
       }'
   ```

Die `autoDeploymentMode`-Eigenschaft ist optional. Wenn Sie es weglassen, ist die Standardeinstellung`Disabled`, und der Endpunkt leitet den Datenverkehr nicht automatisch an neue Revisionen weiter. Notieren Sie sich `revisionId` die Revision in der Antwort.

3. Warten Sie, bis die Revision fertig ist`Active`. *`rev-abc12`*Ersetzen Sie durch die Revisions-ID aus dem vorherigen Schritt.

```
aws lambda-web wait web-function-revision-active \
    --function-name my-web-app \
    --revision-id rev-abc12
```

4. Ruft die Details des Endpunkts ab.

```
aws lambda-web get-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

Wenn der Endpunkt `state` ist`Active`, senden Sie Anfragen an seinen`domainName`. Die Antwort sieht aus wie der folgende Auszug.

```
{
    "endpointName": "my-endpoint",
    "endpointType": "HomeRegion",
    "domainName": "bcdfghjklmnpqrst.b7t.lambda-web.us-east-1.on.aws",
    "authType": "ApplicationManaged",
    "autoDeploymentMode": "LatestRevision",
    "state": "Active",
    "updateStatus": "Successful"
}
```

Um eine Änderung mit den API-Befehlen bereitzustellen (AWS CLI)

- Laden Sie ein neues Paket hoch und erstellen Sie eine Revision. Eine Revision ist ein vollständiger Snapshot. Übergeben Sie die vollständige `buildConfig` Version `serviceConfig` jedes Mal.

```
aws s3 cp ../my-app.zip s3://amzn-s3-demo-bucket/my-app.zip
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
```

```
    --build-config '{
      "codeConfig": {
        "s3Object": { "bucket": "amzn-s3-demo-bucket", "key": "my-app.zip" }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-app-execution-
role"
    }'
```

Da der Endpunkt auf `autoDeploymentMode` eingestellt ist`LatestRevision`, leitet er den gesamten Datenverkehr an die neue Revision weiter, nachdem die Revision geändert wurde`Active`. Wenn Sie `autoDeploymentMode` auf einstellen`Disabled`, leiten Sie den Verkehr stattdessen mit Gewichten `update-web-function-endpoint` und Revisionsgewichten weiter. Weitere Informationen finden Sie unter the section called “Bereitstellungsmuster”.

Bereinigen

Wenn Sie die Webfunktion nicht mehr benötigen, löschen Sie sie:

```
aws lambda-web delete-web-function --function-name my-web-app
```

Mit diesem Befehl werden auch die Endpunkte und Revisionen der Web-Funktion gelöscht. Die Ressourcen, die `deploy` erstellt oder verwendet wurden, werden nicht gelöscht: die IAM-Ausführungsrolle (`awscli-lambdaweb-my-web-app`standardmäßig), die Bereitstellungspakete `lambda-web/my-web-app/` im Amazon S3 S3-Bucket (`awscli-lambdaweb-`*`account-id`*`-`*`region`*`-an`standardmäßig), einschließlich nicht aktueller Objektversionen, oder die Amazon CloudWatch Logs-Protokollgruppen für die Web-Funktion. Löschen Sie diese Ressourcen nur dann separat, wenn Sie sie nicht mehr benötigen. Der verwaltete Amazon S3 S3-Bucket kann von anderen Bereitstellungen in derselben AWS-Konto Region gemeinsam genutzt werden.

Bereitstellen von Webfunktionen

Sie stellen eine Webfunktion bereit, indem Sie Ihren Anwendungscode als ZIP-Paket in einem Amazon S3 S3-Bucket bereitstellen. Wenn Sie eine neue Version bereitstellen, kontrollieren Sie anhand von Bereitstellungsmustern wie Canary und Blue-Green, wie der Datenverkehr zu dieser

Version weitergeleitet wird. Für globale Anwendungen können Sie mehrere Anwendungen AWS-Regionen hinter einem einzigen Endpunkt bereitstellen, und Lambda leitet jede Anfrage basierend auf der Netzwerklatenz an eine Region weiter. Lambda garantiert nicht, dass jede Anfrage an die Region mit der niedrigsten Latenz gesendet wird.

Themen

- Bereitstellen von Web-Funktionscode aus Amazon S3
- Bereitstellungsmuster
- Multi-Region Einsatz

Bereitstellen von Web-Funktionscode aus Amazon S3

Sie stellen Lambda Web Function Code als .zip-Bereitstellungspaket in einem Amazon S3 S3-Bucket bereit. Dies ist die einzige Möglichkeit, Code für eine Webfunktion bereitzustellen. Wenn Sie eine Webfunktion oder Revision erstellen, geben Sie das Objekt in dem `s3Object` Feld an, unter dem `codeConfig` sich die Anforderung befindet. `buildConfig` Geben `s3Object` Sie den Bucket, den Schlüssel und eine optionale versionId ein. Lambda verweist direkt aus Ihrem Bucket auf Ihren Code. Da Lambda jede Revision einer bestimmten Objektversion zuordnet, sind Ihre Bereitstellungen reproduzierbar. Sie können ein einzelnes Bereitstellungspaket für mehrere Konten und Regionen wiederverwenden. AWS

Themen

- Voraussetzungen
- Erforderliche Berechtigungen
- Erstellen einer Webfunktion mit Code aus Amazon S3
- Lebenszyklus eines Objekts
- Multi-Region Bereitstellungen

Voraussetzungen

Bevor Sie Web Function-Code von Amazon S3 bereitstellen, stellen Sie sicher, dass Ihr Bucket und Ihr Bereitstellungspaket die folgenden Anforderungen erfüllen:

S3-Versionierung

Für den Bucket muss die S3-Versionierung aktiviert sein. Lambda verwendet Objektversionen, um zu verfolgen, welche Version Ihres Codes eine Revision verwendet.

Versions-ID

Der `versionId` Parameter ist in der Anfrage optional. Wenn Sie es weglassen, verwendet Lambda die neueste Version des Objekts zum Zeitpunkt der Erstellung. In jedem Fall bindet Lambda die Revision an eine bestimmte Objektversion.

Maximale Größe des Bereitstellungspakets

512 MB (entpackt).

Eintragsdatei

Das Bereitstellungspaket muss die Eintragsdatei Ihrer Anwendung enthalten. Standardmäßig wird die Runtime `index.js` im Stammverzeichnis des Pakets geladen. Um eine andere Eintragsdatei zu verwenden, setzen Sie die `AWS_LAMBDA_NODEJS_ENTRYPOINT` Umgebungsvariable in der Revision auf den Pfad der Datei relativ zum Paketstamm. Nach dem Laden der Eintragsdatei wartet die Runtime darauf, dass Ihr HTTP-Server anfängt, den Host und den Port der `AWS_LAMBDA_HTTP_ENDPOINT` Umgebungsvariablen abzuhören. Wenn die Laufzeitumgebung die Eintragsdatei nicht auflösen kann oder der Server nie mit der Überwachung beginnt, schlägt die Initialisierung fehl und Anfragen an den Endpunkt schlagen fehl.

Lambda überprüft nicht, ob Ihr Server mit der Überwachung beginnt, wenn er einen Endpunkt erstellt oder aktualisiert, um die Revision zu verwenden. Die Revision und der Endpunkt können `Active` in diesen Fällen immer noch angezeigt werden. Senden Sie nach jeder Bereitstellung eine Testanfrage und überprüfen Sie Ihre Anwendungsprotokolle.

Standort des Buckets

Der Bucket kann sich in einem anderen AWS Konto und in einer anderen AWS Region als in der Webfunktion befinden. Der Bucket muss sich in derselben AWS Partition wie die Webfunktion befinden. Der Bucket darf sich nicht in einer Opt-in-Region befinden (einer Region, die standardmäßig nicht aktiviert ist). Wenn eine der Bedingungen nicht erfüllt ist, lehnt Lambda die Anfrage ab. Cross-Region Beim Abrufen von Objekten fallen zusätzliche Amazon S3 S3-Datenübertragungskosten an. Weitere Informationen zu Preisen finden Sie unter Amazon-S3-Preise.

Erforderliche Berechtigungen

Um Webfunktionscode von Amazon S3 aus bereitzustellen, müssen Sie Berechtigungen sowohl für die aufrufende Identität als auch für den Lambda-Serviceprinzipal konfigurieren.

Berechtigungen des Anrufers

Um die Webfunktion oder Revision zu erstellen, benötigt Ihre IAM-Identität die folgenden Berechtigungen für das S3-Objekt:

- `s3:GetObject`
- `s3:GetObjectVersion`

Bei kontoübergreifenden Buckets muss die Bucket-Richtlinie auch der aufrufenden Identität Zugriff auf das Objekt gewähren.

Bucket-Richtlinie für Lambda

Die Bucket-Richtlinie muss dem Lambda-Service Principal (`lambda.amazonaws.com`) `s3:GetObject` und `s3:GetObjectVersion` Berechtigungen gewähren. Definieren Sie für die Richtlinie eine `aws:SourceArn` Bedingung, die dem ARN-Muster der Revision entspricht. Verwenden Sie es `ArnLike` mit einem Platzhalter für die Revisions-ID, da Lambda die Revisions-ID generiert, wenn Sie die Revision erstellen.

Wenn das Objekt mit einem AWS KMS Schlüssel (SSE-KMS) verschlüsselt ist, muss die Schlüsselrichtlinie auch den Aufruf `kms:Decrypt` dieses Schlüssels `lambda.amazonaws.com` ermöglichen. Lambda überprüft den Zugriff des Service Principal, wenn Sie die Revision erstellen, und lehnt die Anfrage ab, wenn der Zugriff fehlt.

Die folgende Beispiel-Bucket-Richtlinie gewährt Lambda Zugriff zum Lesen Ihrer Bereitstellungspakete für jede Version der angegebenen Webfunktion.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowLambdaWebFunctionCodeAccess",
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
```

```
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": "arn:aws:s3:::my-bucket/*",
            "Condition": {
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:lambda:us-east-1:111122223333:web-
function/my-function/revision/*"
                }
            }
        }
    ]
}
```

Um den Zugriff auf das key prefix zu beschränken, das Ihre Bereitstellungspakete enthält, ändern Sie den `Resource` *my-bucket* Wert in`arn:aws:s3:::`*my-bucket*`/`*my-prefix*`/*`, wobei Ihr Bucket-Name und *my-prefix* das key prefix sind. Behalten Sie das `/*` Ende des Präfixes bei. Ändern Sie den `revision/*` Platzhalter nicht in`aws:SourceArn`.

Erstellen einer Webfunktion mit Code aus Amazon S3

Sie geben Ihr S3-Bereitstellungspaket im `codeConfig` Abschnitt von an`buildConfig`. Die folgenden Verfahren zeigen, wie Sie eine Webfunktion erstellen und wie Sie mit dem eine neue Revision erstellen AWS CLI.

Um eine Webfunktion mit Code aus Amazon S3 zu erstellen (AWS CLI)

- Führen Sie den folgenden Befehl aus, um eine Webfunktion zu erstellen, die auf Ihr ZIP-Bereitstellungspaket in Amazon S3 verweist.

```
aws lambda-web create-web-function \
    --function-name my-web-app \
    --revision-config '{
      "buildConfig": {
        "codeConfig": {
          "s3Object": { "bucket": "my-bucket", "key": "my-app.zip" }
        },
        "runtimeConfig": { "runtime": "nodejs24.x" }
      },
      "serviceConfig": {
        "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role"
```

```
      }
    }' \
    --endpoint-config '{
      "endpointName": "my-endpoint",
      "endpointType": "HomeRegion",
      "authType": "ApplicationManaged"
    }'
```

Wenn Sie es weglassen`versionId`, verwendet Lambda die neueste Version des Objekts zum Zeitpunkt der Erstellung.

Um eine neue Revision mit einer bestimmten Objektversion zu erstellen (AWS CLI)

- Führen Sie den folgenden Befehl aus, um eine neue Revision zu erstellen, die eine bestimmte Version Ihres Bereitstellungspakets verwendet.

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --build-config '{
      "codeConfig": {
        "s3Object": {
          "bucket": "my-bucket",
          "key": "my-app.zip",
          "versionId": "abc123def456"
        }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role"
    }'
```

Durch die Angabe `versionId` binden Sie die Revision an eine exakte Version des Objekts und nicht an die neueste Version.

Lebenszyklus eines Objekts

> **Löschen Sie keine S3-Objekte und entfernen Sie den Lambda-Zugriff für aktive Revisionen nicht**
>
> Lambda verweist auf Ihren Code aus Ihrem Amazon S3 S3-Bucket. Wenn Sie das S3-Objekt löschen, das eine Revision verwendet, oder die spezifische Objektversion, die die Revision verwendet, wechselt die Revision in den `Failed` Zustand, in dem Lambda das nächste Mal den Zugriff auf das Objekt überprüft, und kann dann keinen Datenverkehr mehr bereitstellen. Behalten Sie die Objektversionen für alle Revisionen bei, die Sie noch verwenden.
> Das Entfernen des Zugriffs des Lambda-Serviceprinzipals auf den Bucket hat den gleichen Effekt. Wenn Sie die Bucket-Richtlinie so ändern, dass sie die `s3:GetObjectVersion` Berechtigungen verweigert oder entfernt `s3:GetObjectlambda.amazonaws.com`, werden bestehende Revisionen, die auf Objekte in diesem Bucket verweisen, in den `Failed` Status versetzt. Behalten Sie sowohl die Objekte als auch den Lesezugriff des Lambda-Serviceprinzipals bei, solange eine Revision auf diese Objekte verweist.
> Der `Failed` Zustand ist dauerhaft. Durch das Wiederherstellen des gelöschten Objekts oder der Bucket-Berechtigungen wird der `Active` Status einer Version nicht wiederhergestellt. Erstellen Sie zur Wiederherstellung eine neue Revision, die auf eine verfügbare Objektversion verweist, und aktualisieren Sie dann die Revisionsgewichte Ihrer Endpunkte, um den Datenverkehr dorthin weiterzuleiten. Informationen zu Bereitstellungsmustern finden Sie unter. the section called “Bereitstellungsmuster”

Lambda verlagert den Verkehr nicht automatisch von einer `Failed` Revision weg. Anfragen, die ein Endpunkt an eine `Failed` Revision weiterleitet, werden mit dem HTTP-Statuscode 409 und dem `lambda-web-error-type` Antwortheader von `Platform.RevisionNotActive` abgelehnt. Wenn der Endpunkt den Datenverkehr auf mehrere Revisionen verteilt, ist nur der Anteil der Anfragen betroffen, die an die `Failed` Revision weitergeleitet wurden. Anfragen, die an fehlerfreie Versionen weitergeleitet werden, werden normal bearbeitet. Um die Ausfälle zu verhindern, aktualisieren Sie die Revisionsgewichte des Endpunkts, um die Revision zu entfernen. `Failed`

Wenn Sie das Objekt mit einem neuen Upload überschreiben, wird eine neue S3-Version erstellt. Dies wirkt sich nicht auf bestehende Revisionen aus: Wenn Sie eine Revision erstellen, löst Lambda die genaue Objektversion auf und pinnt sie an, auch wenn Sie sie weglassen. `versionId` Wenn Sie jedoch eine alte Objektversion löschen, auf die eine Revision verweist, wechselt diese Revision

in den Status. `Failed` Das Hochladen eines Ersatzobjekts hilft nicht, da es sich bei dem Ersatz um eine neue Version mit einer anderen Versions-ID handelt.

Multi-Region Bereitstellungen

Für Webfunktionen mit `MultiRegion` oder `PerRegion` Endpunkten liest Lambda Ihr Quellobjekt aus dem von Ihnen angegebenen Bucket, unabhängig davon, in welcher Region sich der Bucket befindet. Lambda repliziert dann das vorbereitete Codeartefakt, das es von diesem Objekt ableitet, in die anderen Regionen. Sie müssen Ihr Bereitstellungspaket nicht in Buckets in anderen Regionen kopieren. Weitere Informationen darüber, was Lambda repliziert, finden Sie unter. the section called “Was Lambda repliziert”

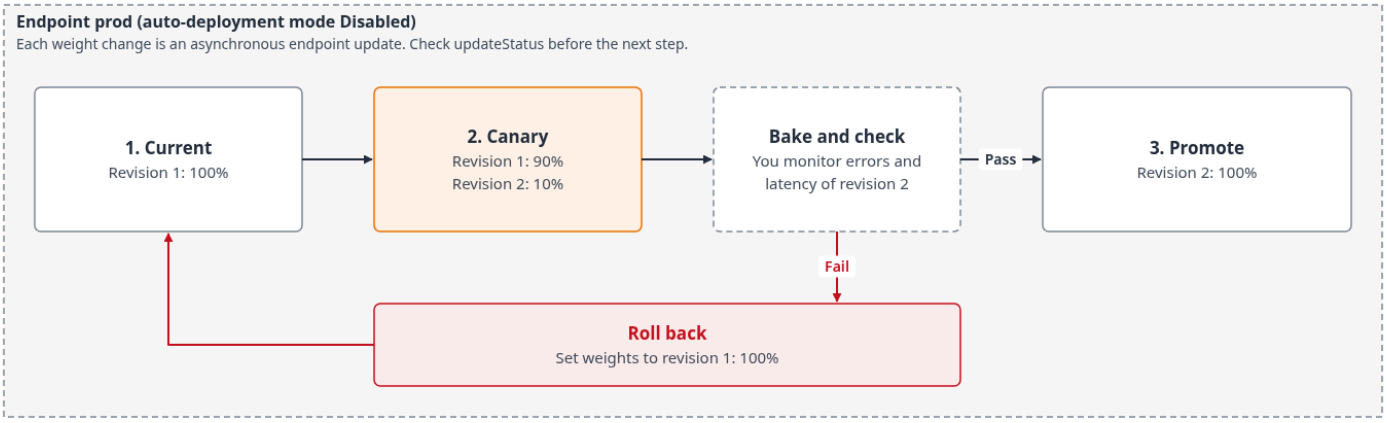
Bereitstellungsmuster

Um eine neue Version Ihrer AWS Lambda Web Function-Anwendung zu veröffentlichen, erstellen Sie eine neue Revision. Endgeräte, die so `autoDeploymentMode` eingestellt sind, dass sie den gesamten LatestRevisionDatenverkehr dorthin weiterleiten, sobald sie aktiv werden. Endgeräte, für die die Option Deaktiviert aktiviert ist, `autoDeploymentMode` stellen weiterhin ihre aktuellen Versionen bereit, bis Sie die Gewichtung der Versionen aktualisieren. Mithilfe des automatischen Bereitstellungsmodus und der Revisionsgewichte des Endpunkts steuern Sie, wie der Datenverkehr von der aktuellen zur neuen Version übertragen wird. In den folgenden Abschnitten werden beide Modi für die automatische Bereitstellung, gängige Bereitstellungsmuster (Canary, Blue-Green und Rollback) und die Regeln für die Gewichtung von Revisionen erläutert.

Änderungen an `autoDeploymentMode` und `revisionWeights` sind asynchrone Endpunkt-Updates. Warten Sie`updateStatus`, bis das Datum erreicht ist, `Successful` bevor Sie die Bereitstellung evaluieren oder ein weiteres Endpunkt-Update einreichen. Falls ein Update fehlschlägt, `updateStatusReason` wird die Ursache beschrieben.

Das folgende Diagramm zeigt eine Canary-Bereitstellung auf einem Endpunkt, dessen automatischer Bereitstellungsmodus Deaktiviert ist. Jede Gewichtsänderung ist ein asynchrones Endpunkt-Update.

1. Revision 1 beginnt mit 100 Prozent des Datenverkehrs.
2. Sie leiten 10 Prozent des Datenverkehrs an Version 2 weiter und überwachen ihn.
3. Wenn Revision 2 Ihre Prüfungen bestanden hat, leiten Sie den gesamten Datenverkehr an sie weiter.
4. Wenn Revision 2 fehlschlägt, setzen Sie die Gewichtung auf 100 Prozent zurück auf Version 1.



Rolling back is another endpoint update, not an instant switch. Keep the earlier revision until the new one is stable.

Auto-deployment Modi

Die `autoDeploymentMode` Einstellung auf einem Endpunkt steuert, ob Lambda den Datenverkehr bei der Bereitstellung automatisch an neue Versionen weiterleitet. Sie legen dies fest, wenn Sie den Endpunkt erstellen, und Sie können es auf einem vorhandenen Endpunkt ändern.

Wenn Sie es weglassen`autoDeploymentMode`, ist es standardmäßig. `Disabled` `LatestRevision`wird nur für `HomeRegion` Endgeräte unterstützt, daher verwenden `PerRegion` Endpunkte `MultiRegion` immer. `Disabled` Bei der Verwendung `Disabled` legen Sie `revisionWeights` den Endpunkt fest, wie unter beschrieben. the section called “Revisionsgewichte”

LatestRevision

> Jedes Mal, wenn eine neue Revision aktiv wird, leitet Lambda 100% des Datenverkehrs des Endpunkts an sie weiter. Sie müssen die Revisionsgewichte nicht aktualisieren, und Sie können keine Revisionsgewichte festlegen, solange sich der Endpunkt in diesem Modus befindet. Verwenden Sie diesen Modus für Entwicklungs- und Staging-Umgebungen, in denen Sie eine automatische Bereitstellung ohne manuelles Eingreifen wünschen.

Deaktiviert

> Lambda ändert das Traffic-Routing nicht, wenn Sie eine neue Revision erstellen. Sie steuern die Weiterleitung des Datenverkehrs manuell, indem Sie die Gewichtung der Revisionen festlegen. Verwenden Sie diesen Modus für Produktionsumgebungen, in denen Sie die Einführung neuer Versionen mit kanarischen oder blaugrünen Bereitstellungsmustern steuern möchten.

Um den Modus für die automatische Bereitstellung zu ändern (AWS CLI)

1. Verwenden Sie AWS Command Line Interface (AWS CLI), um den Endpunkt zu aktualisieren und die automatische Bereitstellung zu deaktivieren.

```
aws lambda-web update-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint \
    --auto-deployment-mode Disabled
```

2. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

```
aws lambda-web wait web-function-endpoint-updated \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

Wechseln zwischen den Modi

Wenn Sie ohne Angabe von Versionsgewichten von Deaktiviert LatestRevisionzu wechseln, bleibt das aktuelle Datenverkehrsrouting nach erfolgreicher Aktualisierung unverändert. Die Version, an die der Endpunkt den Verkehr vor dem Update weitergeleitet hat, empfängt weiterhin 100% des Datenverkehrs, auch wenn Sie neuere Versionen erstellt haben. Dieser Datenverkehr wird beibehalten, bis Sie die Revisionsgewichte aktualisieren. Sie können in derselben `UpdateWebFunctionEndpoint` Anfrage auch den Modus wechseln und die Revisionsgewichte festlegen.

Wenn Sie von Deaktiviert zu wechseln LatestRevision, aktualisiert Lambda den Endpunkt so, dass 100% des Datenverkehrs auf die neueste Version weitergeleitet werden. Die neueste Version muss aktiv sein. Wenn sie „Ausstehend“ oder „Fehlgeschlagen“ lautet, schlägt die Anforderung mit einem Konfliktfehler fehl. Nach erfolgreicher Aktualisierung gelten die zuvor konfigurierten Revisionsgewichte nicht mehr.

Revisionsgewichte

Wenn der automatische Bereitstellungsmodus deaktiviert ist, steuern Sie die Verkehrsweiterleitung, indem Sie Revisionen Gewichte zuweisen. Lambda verteilt den Traffic zwischen den Revisionen proportional zu ihren Gewichten.

Bei der Gewichtung von Revisionen handelt es sich um eine Einstellung am Endpunkt (die `revisionWeights` Eigenschaft), mit der Sie Änderungen vornehmen. `UpdateWebFunctionEndpoint` Sie sind keine Einstellung für die Revision.

Für die Gewichte der Revisionen gelten die folgenden Regeln:

- Jedes Gewicht ist eine Ganzzahl von 1 bis 100.
- Die Gewichte in einer Anfrage müssen sich auf genau 100 summieren.
- Jede Revision kann in den Gewichtungen nur einmal vorkommen.
- Jede Version der Gewichtungen muss sich beim Einstellen der Gewichtungen im Status Aktiv befinden. Andernfalls lehnt die API die Anfrage ab.
- Ein Endpunkt leitet den Datenverkehr gleichzeitig an maximal zwei Revisionen weiter.
- Keine Gewichtung hat einen Standardwert. Um den gesamten Datenverkehr an eine Revision weiterzuleiten, geben Sie nur die Revision mit einer Gewichtung von 100 an. Eine Revision, die nicht in den Gewichtungen enthalten ist, empfängt keinen Verkehr von diesem Endpunkt.

Revisionsbezeichner werden von Lambda generiert, wenn Sie eine Revision erstellen. Verwenden Sie den folgenden Befehl, um Revisionsbezeichner für Ihre Web-Funktion zu finden:

```
aws lambda-web list-web-function-revisions \
    --function-name my-web-app
```

In der Antwort werden die Revisionen in der Reihenfolge von der neuesten zur ältesten Version aufgeführt, einschließlich ihrer Kennungen, Erstellungszeitstempel und des aktuellen Status.

Bereitstellungsmuster

Die folgenden Bereitstellungsmuster verwenden Revisionsgewichte, um zu steuern, wie der Datenverkehr von Ihrer aktuellen Revision zu einer neuen Revision fließt. Für diese Muster muss der automatische Bereitstellungsmodus des Endpunkts (die `autoDeploymentMode` Endpunkteinstellung) deaktiviert sein. Der Endpunkt, den der `aws lambda-web deploy` Befehl erstellt, verwendet LatestRevision. Um diese Muster auf diesem Endpunkt zu verwenden, schalten Sie ihn zunächst auf Deaktiviert um, wie unter gezeigtthe section called “Auto-deployment Modi”.

Bereitstellung auf Canary

Bei einer Bereitstellung auf Canary wird ein kleiner Teil des Datenverkehrs an die neue Version weitergeleitet, sodass Sie zunächst nach Fehlern suchen können, bevor Sie den gesamten

Datenverkehr weiterleiten. Wenn bei der neuen Version Probleme auftreten, ist nur ein kleiner Teil der Anfragen betroffen.

Um eine Canary-Bereitstellung durchzuführen (AWS CLI)

1. Erstellen Sie eine neue Version mit Ihrem aktualisierten Code.

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --build-config '{
      "codeConfig": {
        "s3Object": { "bucket": "my-bucket", "key": "my-app-v2.zip" }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role"
    }'
```

   Notieren Sie sich die Revisions-ID in der Antwort.

   Eine neue Revision kopiert keine Einstellungen aus der aktuellen Revision. Jede `serviceConfig` Einstellung, die Sie auslassen, erhält ihren Standardwert, z. B. einen Wert `timeoutSeconds` von 30 und keine Umgebungsvariablen. Schließen Sie alle Einstellungen ein, die die aktuelle Version verwendet, z. B.`environmentVariables`, `timeoutSecondsmaxConcurrencyPerEnvironment`, und`telemetryConfig`. Um die Konfiguration der aktuellen Revision zu sehen, führen Sie den Befehl aus`aws lambda-web get-web-function-revision`.

2. Warten Sie, bis die neue Revision aktiv ist. Die API lehnt Revisionsgewichte ab, die auf eine Revision verweisen, die nicht aktiv ist.

```
aws lambda-web wait web-function-revision-active \
    --function-name my-web-app \
    --revision-id rev-def45
```

3. Leitet 10% des Datenverkehrs an die neue Revision und 90% an die aktuelle Revision weiter.

```
aws lambda-web update-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint \
```

```
    --revision-weights '[{"revisionId": "rev-abc12", "weight": 90}, {"revisionId":
 "rev-def45", "weight": 10}]'
```

4. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

```
aws lambda-web wait web-function-endpoint-updated \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

5. Überwachen Sie die neue Version anhand von Anwendungsprotokollen oder benutzerdefinierten Metriken. Fügen Sie eine Build- oder Revisions-ID hinzu, die Sie kontrollieren. Metriken für Webfunktionen enthalten keine Revisionsdimension. Daher können sie Canary-Fehler nicht von Fehlern in der aktuellen Version trennen. Details hierzu finden Sie unter the section called “Metriken”. Wenn der Canary die Fehler- und Latenzschwellenwerte erfüllt, die Sie vor dem Rollout definiert haben, erhöhen Sie seine Gewichtung.

```
aws lambda-web update-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint \
    --revision-weights '[{"revisionId": "rev-abc12", "weight": 50}, {"revisionId":
 "rev-def45", "weight": 50}]'
```

6. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

```
aws lambda-web wait web-function-endpoint-updated \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

7. Wenn Sie sicher sind, dass die neue Version fehlerfrei ist, leiten Sie den gesamten Datenverkehr an sie weiter.

```
aws lambda-web update-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint \
    --revision-weights '[{"revisionId": "rev-def45", "weight": 100}]'
```

8. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

```
aws lambda-web wait web-function-endpoint-updated \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

Blue-green Bereitstellung

Bei einer blaugrünen Bereitstellung wird der Datenverkehr weiterhin von der aktuellen Version bereitgestellt, während Sie die neue Version validieren. Anschließend wird der gesamte Datenverkehr in einem Endpunkt-Update auf die neue Version umgestellt. Das folgende Verfahren validiert mit einer gleichmäßigen Verteilung des Datenverkehrs. Verwenden Sie zur Validierung ohne Produktionsdatenverkehr einen separaten Testendpunkt, wie unter beschriebenthe section called "Überlegungen zur Bereitstellung".

Um eine blaugrüne Bereitstellung durchzuführen (AWS CLI)

1. Erstellen Sie eine neue Version mit Ihrem aktualisierten Code und warten Sie, bis sie aktiv ist. Verwenden Sie dieselben Befehle wie im vorherigen Canary-Bereitstellungsverfahren.
2. Teilen Sie den Verkehr gleichmäßig auf die beiden Revisionen auf.

```
aws lambda-web update-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint \
    --revision-weights '[{"revisionId": "rev-abc12", "weight": 50}, {"revisionId":
 "rev-def45", "weight": 50}]'
```

3. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

```
aws lambda-web wait web-function-endpoint-updated \
    --function-name my-web-app \
    --endpoint-name my-endpoint
```

4. Überwachen Sie beide Versionen anhand von Anwendungsprotokollen oder benutzerdefinierten Metriken. Fügen Sie eine Build- oder Revisions-ID hinzu, die Sie kontrollieren. Wenn die neue Version die Integritätskriterien erfüllt, die Sie vor dem Rollout definiert haben, leiten Sie den gesamten Datenverkehr an sie weiter.

```
aws lambda-web update-web-function-endpoint \
    --function-name my-web-app \
    --endpoint-name my-endpoint \
    --revision-weights '[{"revisionId": "rev-def45", "weight": 100}]'
```

5. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

```
aws lambda-web wait web-function-endpoint-updated \
    --function-name my-web-app \
```

```
    --endpoint-name my-endpoint
```

Wird zurückgesetzt

Wenn Sie Probleme mit einer neuen Version feststellen (hohe Fehlerraten, erhöhte Latenz), können Sie ein Rollback mit einem einzigen Endpunkt-Update durchführen, das den gesamten Datenverkehr an die vorherige, zweifelsfrei funktionierende Version weiterleitet.

> **Rollback im Modus LatestRevision**
>
> Der automatische Bereitstellungsmodus eines Endpunkts ist zwar aktiviert, aber eine Anfrage LatestRevision, die nur Revisionsgewichte festlegt, schlägt fehl. Mit dem folgenden Verfahren wird der automatische Bereitstellungsmodus auf Deaktiviert gesetzt und die Revision wird in derselben Anfrage gewichtet, sodass er in beiden Modi funktioniert.
> Der Endpunkt bleibt dann im deaktivierten Modus, bis Sie ihn ändern.

Um zu einer früheren Version zurückzukehren (AWS CLI)

1. Aktualisieren Sie den Endpunkt so, dass er 100% des Datenverkehrs an die zweifelsfrei funktionierende Version weiterleitet, und setzen Sie den Modus für die automatische Bereitstellung auf Deaktiviert.

   ```
   aws lambda-web update-web-function-endpoint \
       --function-name my-web-app \
       --endpoint-name my-endpoint \
       --auto-deployment-mode Disabled \
       --revision-weights '[{"revisionId": "rev-abc12", "weight": 100}]'
   ```

2. Warten Sie, bis das Endpunkt-Update abgeschlossen ist.

   ```
   aws lambda-web wait web-function-endpoint-updated \
       --function-name my-web-app \
       --endpoint-name my-endpoint
   ```

Nachdem das Update erfolgreich war, beendet Lambda die Weiterleitung neuer Anfragen an die problematische Revision. In-flight Anfragen zu dieser Revision werden normal abgeschlossen.

Weitere Hinweise zum Aktualisierungsstatus von Endpunkten finden Sie unterthe section called “Endpunktzustände”.

Wenn für die Revision, auf die Sie ein Rollback durchführen, nur wenige oder gar keine Ausführungsumgebungen laufen, skaliert Lambda sie, wenn der Verkehr dorthin verlagert wird. Anfragen können warten, bis die Ausführungsumgebungen initialisiert werden, was die Latenz erhöht. Anfragen können aufgrund von Startfehlern, der Ratenbeschränkung oder maximalen Umgebungen des Endpunkts oder des vCPU-Kontingents des Kontos immer noch fehlschlagen. Bei einigen dieser Fehler werden HTTP 429-Antworten zurückgegeben.

Überlegungen zur Bereitstellung

Was löst eine neue Revision aus?

Lambda erstellt eine neue Revision, wenn Sie aufrufen`CreateWebFunctionRevision`. Dazu gehören die Bereitstellung von neuem Anwendungscode und das Ändern von Konfigurationseinstellungen wie Timeout, Parallelität oder Umgebungsvariablen. Jede Änderung führt zu einer unveränderlichen Revision.

Kann ich eine neue Version testen, bevor ich den Produktionsdatenverkehr an sie weiterleite?

Ja. Sie können eine Revision nur über einen Endpunkt aufrufen, es gibt also zwei Ansätze:

- Dedizierter Testendpunkt — Erstellen Sie einen separaten Endpunkt für Tests. Leiten Sie den Datenverkehr zuerst an die neue Version auf dem Testendpunkt weiter (indem Sie die Revisionsgewichte des Testendpunkts festlegen), validieren Sie sie und aktualisieren Sie dann die Gewichte des Produktionsendpunkts, sodass sie die neue Version enthalten.
- Minimale Gewichtung — Weisen Sie der neuen Revision eine Gewichtung von 1% auf dem Produktionsendpunkt zu. Dadurch wird ein kleiner Teil des Produktionsverkehrs an die neue Version weitergeleitet, sodass Sie Fehler mit minimaler Auswirkung auf den Kunden überwachen können.

Was passiert mit Revisionen, die nicht in den Gewichten der Revisionen enthalten sind?

Eine Revision, die nicht in den Revisionsgewichten eines Endpunkts enthalten ist, empfängt keinen Datenverkehr von diesem Endpunkt. Ein anderer Endpunkt derselben Webfunktion kann immer noch zu ihm weiterleiten. Die Revision bleibt aktiv, bis Sie sie löschen oder bis sie bei der automatischen Bereinigung entfernt wird. Es wechselt zu Fehlgeschlagen, wenn Lambda den Zugriff auf sein Amazon S3 S3-Objekt verliert. Informationen zur automatischen Bereinigung finden Sie unter. the section called “Revisionsgrenzen und automatische Bereinigung”

Wie finde ich heraus, welche Version die neueste ist?

Rufen Sie die Seite `ListWebFunctionRevisions` auf. In der Antwort werden die Versionen in absteigender Reihenfolge von aufgeführt`createdAt`, sodass die erste Revision in der Liste die neueste ist.

Kann ich einen Endpunkt löschen, während er Datenverkehr weiterleitet?

Ja. Wenn Sie einen Endpunkt löschen, beendet Lambda die Weiterleitung neuer Anfragen an diesen Endpunkt, und Anfragen an den Domainnamen des Endpunkts werden zurückgewiesen, nachdem der Löschvorgang wirksam wird. Die Revisionen selbst sind davon nicht betroffen und können weiterhin von anderen Endpunkten verwendet werden.

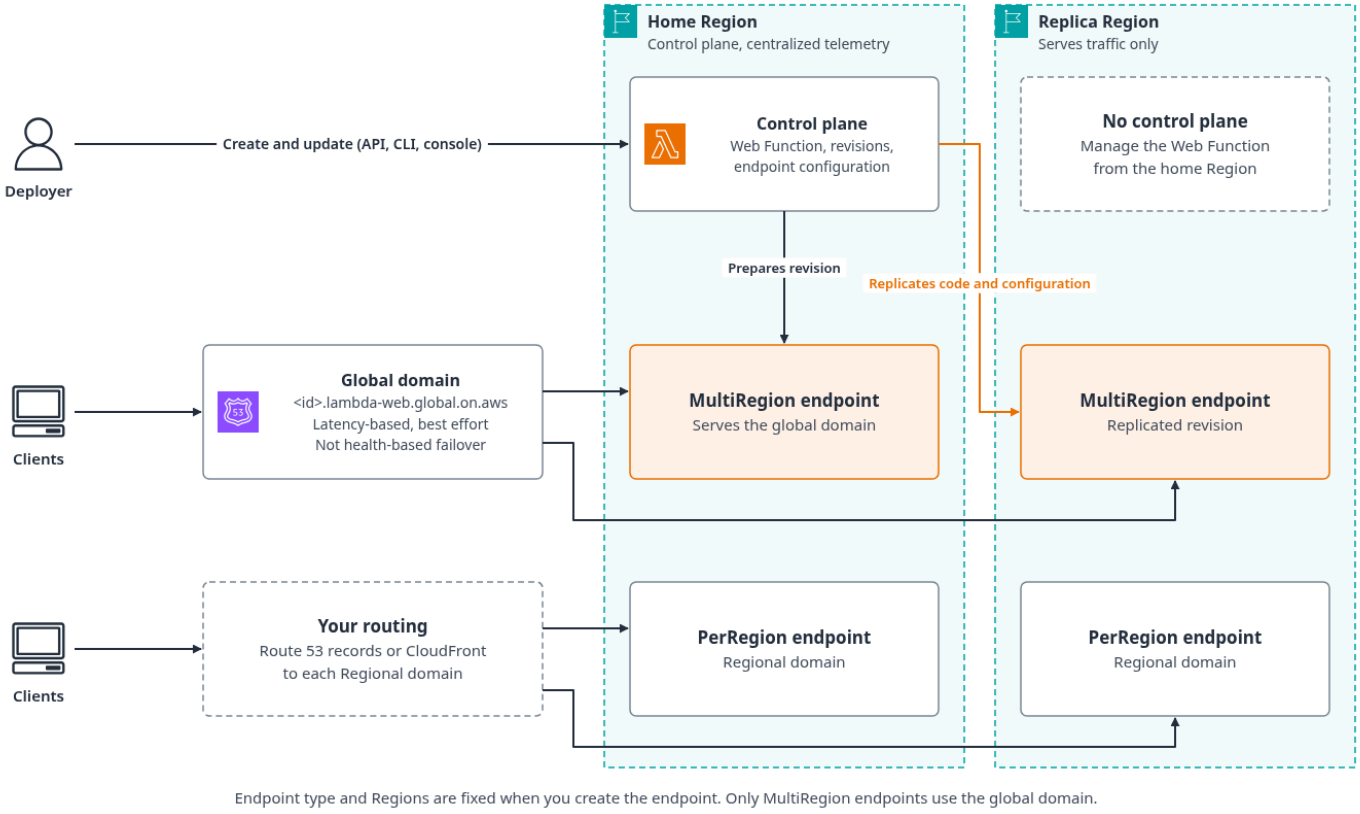
Kann ich eine Revision löschen, während sie Traffic empfängt?

Nein. Wenn ein Endpunkt Traffic an die Revision weiterleitet, auch wenn die `regionalEndpoints` Eigenschaft eines `MultiRegion` oder `PerRegion` -Endpunkts auf die Revision verweist, schlägt die `DeleteWebFunctionRevision` Anfrage mit einem Konfliktfehler fehl. Um die Revision zu löschen, aktualisieren Sie zunächst die Revisionsgewichte des Endpunkts, um ihn aus dem Datenverkehrsrouting auf allen Endpunkten zu entfernen, und löschen Sie ihn dann. Da Lambda das Löschen blockiert, solange einer Revision eine Gewichtung zugewiesen wurde, ändert das Löschen einer Revision niemals die Verkehrsverteilung eines Endpunkts.

Multi-Region Einsatz

Für MultiRegionPerRegionEndgeräte repliziert Lambda Laufzeitartefakte in alle Zielregionen, sodass Ihre Anwendung den Datenverkehr in der Nähe von Endbenutzern bereitstellen kann. Sie verwalten die Webfunktion vollständig von der Heimatregion aus. Replizierte Ressourcen sind in Replikatregionen weder sichtbar noch änderbar.

Das folgende Diagramm zeigt, wie die Steuerungsebene in der Heimatregion eine Revision vorbereitet und deren Code und Konfiguration in eine Replikatregion repliziert, und zeigt, wie Clients Endpunkte erreichen. MultiRegionPerRegion Eine Replikatregion hat keine Steuerungsebene. Ein MultiRegionEndpunkt hat eine globale Domäne, die nach Latenz und nicht nach dem Zustand der Region weiterleitet. Für einen PerRegionEndpunkt verwalten Sie, wie Clients die regionale Domäne in jeder Region erreichen, z. B. mit Route 53-Datensätzen oder CloudFront. Sie legen den Endpunkttyp und die Regionen fest, wenn Sie den Endpunkt erstellen, und Sie können sie später nicht mehr ändern.



Themen

- Was Lambda repliziert
- Sichtbarkeit von Replikaten
- Wenn die Replikation stattfindet
- Funktionsweise der Weiterleitung
- Replikationsfehler
- Letztendliche Datenkonsistenz

Was Lambda repliziert

Lambda repliziert Folgendes in jede Zielregion:

- Vorbereitetes Codeartefakt — Das interne Laufzeitartefakt, das Lambda aus dem versionierten Amazon S3 S3-Quellobjekt ableitet.
- Revisionskonfiguration — Timeout, Parallelität, Umgebungsvariablen und Laufzeiteinstellungen.

- Resource-based Richtlinien — Mit der Webfunktion verknüpfte Richtlinien, die Zugriff auf die Webfunktion gewähren.
- Tags — Die Tags der Webfunktion.

Kontingente für Webfunktionen gelten pro AWS-Region. Jede Zielregion erzwingt ihre eigenen Kontokontingente auf dem replizierten Endpunkt. Beispielsweise ist die Skalierung der Ausführungsumgebung in einer Replikatregion durch das vCPU-Kontingent dieser Region begrenzt. Informationen zu den Kontenkontingenten finden Sie unter. the section called “Kontingente für Konten”

Sichtbarkeit von Replikaten

Replizierte Ressourcen sind in Replikatregionen nicht sichtbar. Die Steuerungsebene für eine Webfunktion wird nur in ihrer Heimatregion bedient, sodass alle API-Anfragen an die Heimatregion gesendet werden müssen. Dies umfasst:

- Listet Anfragen für die Webfunktion, ihre Versionen und ihre Endpunkte auf, ruft sie ab und beschreibt sie.
- Erstellen, aktualisieren und löschen Sie Anfragen für die Funktion, die Revisionen und die Endpunktkonfiguration.

Eine API-Anfrage, die an den Endpunkt der Kontrollebene einer Replikatregion gesendet wird, schlägt fehl, unabhängig davon, wo sich der Aufrufer befindet. Die replizierten Laufzeitartefakte existieren nur für die Datenebene, um den Datenverkehr zu verarbeiten.

Wenn die Replikation stattfindet

Die Replikation wird ausgelöst, wenn Sie einen PerRegionOR-Endpunkt erstellen MultiRegionoder aktualisieren. Insbesondere findet die Replikation statt, wenn:

- Sie erstellen einen MultiRegion PerRegion OR-Endpunkt, dessen Revisionsgewichte auf eine Revision verweisen.
- Sie aktualisieren die Revisionsgewichte an einem MultiRegion PerRegion OR-Endpunkt, um eine neue Revision einzubeziehen.
- Sie aktualisieren andere Endpunkteinstellungen, z. B. `authTypescalingConfig`, oder`throttleConfig`, auf einem MultiRegion PerRegion OR-Endpunkt.

Das Erstellen einer Revision allein löst keine Replikation aus. Die Revision muss in den Revisionsgewichten eines Endpunkts enthalten sein, für den Zielregionen konfiguriert sind. Da für MultiRegion PerRegion Endpoints nur der Modus Deaktivierte automatische Bereitstellung verfügbar ist, ist die Replikation immer das Ergebnis einer von Ihnen vorgenommenen Endpunktänderung.

Funktionsweise der Weiterleitung

Das Routing-Verhalten hängt vom Endpunkttyp ab:

MultiRegion

Lambda bietet einen einzigen globalen Domainnamen. Anfragen werden mithilfe von latenzbasiertem Routing weitergeleitet. Anfragen werden an die Region weitergeleitet, die in der Regel die niedrigste Netzwerklatenz für den Anrufer bietet, basierend auf Latenzmessungen, die Route 53 im Laufe der Zeit sammelt. Da sich die Internetlatenz ändert, garantiert Lambda nicht, dass jede Anfrage an die Region mit der niedrigsten Latenz gesendet wird. Lambda verwaltet automatisch Routen für Sie.

PerRegion

Lambda stellt in jeder Zielregion einen separaten regionalen Domainnamen bereit. Sie sind dafür verantwortlich, zu verwalten, wie der Traffic jede regionale Domain erreicht. Sie können beispielsweise Ihre eigene Route 53-Routing-Richtlinie konfigurieren oder eine CloudFront Verteilung vor den regionalen Domänen platzieren.

Replikationsfehler

Die Replikation kann in einer Region aufgrund eines vorübergehenden Dienstproblems fehlschlagen. Wenn Sie Ihre Webfunktion mit einem vom Kunden verwalteten Schlüssel verschlüsseln, führt ein fehlendes Schlüsselreplikat in der Zielregion ebenfalls dazu, dass die Replikation fehlschlägt. Informationen zu den wichtigsten Anforderungen für mehrere Regionen finden Sie unter. the section called “Konfiguration eines vom Kunden verwalteten Schlüssels”

Die Auswirkungen eines Replikationsfehlers hängen davon ab, ob Sie den Endpunkt erstellen oder aktualisieren. Wenn die Replikation in einer Region fehlschlägt, während Lambda den Endpunkt erstellt, wechselt der Endpunkt in den Status Fehlgeschlagen und kann in keiner Region aufgerufen werden. Um es erneut zu versuchen, löschen Sie den Endpunkt und erstellen Sie ihn erneut. Wenn die Replikation in einer Region während eines Endpunkt-Updates fehlschlägt, ist der Fehler auf diese Region beschränkt. Der Endpunkt bleibt aufrufbar und die anderen Regionen funktionieren weiterhin normal.

Wenn die Replikation einer neuen Version in einer Region fehlschlägt, stellt diese Region weiterhin Datenverkehr mit der vorherigen Version bereit, die erfolgreich bereitgestellt wurde. Der Endpunkt führt kein Rollback durch. Er arbeitet weiterhin mit dem letzten als funktionierend bekannten Status in jeder Region.

Sie können den Status jeder Region einsehen und den Status im `regionalEndpoints` `GetWebFunctionEndpoint` Antwortfeld aktualisieren. Der Eintrag jeder Region beinhaltet:

- `state`und `stateReason` — Der Bereitstellungsstatus der Region während der Endpunkterstellung.
- `updateStatus`und `updateStatusReason` — Das Ergebnis des letzten Updates (z. B. einer Revisionsbereitstellung) für diese Region.

Um einen Replikationsfehler zu beheben, beheben Sie die Ursache (erstellen Sie beispielsweise ein AWS KMS Schlüsselreplikat in der betroffenen Region). Rufen `UpdateWebFunctionEndpoint` Sie erneut auf, um ein fehlgeschlagenes Endpunkt-Update zu wiederholen, nachdem Sie die Ursache behoben haben. Sie können dieselben Einstellungen wiederholen oder nur den Funktions- und Endpunktnamen angeben. Bei beiden Anfragen wird eine neue Bereitstellung in den Zielregionen gestartet. Dies funktioniert nur, solange der Endpunkt aktiv ist. Wenn die Endpunkterstellung fehlgeschlagen ist, löschen Sie den Endpunkt und erstellen Sie ihn erneut.

Letztendliche Datenkonsistenz

Die Replikation von Laufzeitartefakten ist letztendlich konsistent. Gehen Sie nicht davon aus, dass jede Region die neue Konfiguration ausführt, bis jede Region `updateStatus` in `regionalEndpoints` Berichten aufgeführt ist`Successful`.

Resource-based Nach Code- und Konfigurationsänderungen kann die Übertragung von Richtlinienänderungen beendet werden. Während des Replikationsfensters werden Anfragen in Replikatregionen möglicherweise mit der vorherigen Version der Richtlinie verglichen. Weitere Informationen finden Sie unter the section called “Replikation und letztendliche Konsistenz”.

HTTP-Anfragen und -Antworten

Ein Webfunktions-Endpunkt akzeptiert HTTP/1.1 Anfragen und leitet sie an Ihre Anwendung weiter. HTTP/1.1 Lambda lehnt HTTP/1.0 Anfragen mit HTTP 505 ab (HTTP-Version wird nicht unterstützt). Lambda behält die Anforderungsmethode, den Pfad, die Abfragezeichenfolge, den Hauptteil und die meisten Header bei. Es entfernt oder ändert Header, die die Verbindung zwischen dem Client und

dem Endpunkt beschreiben. Außerdem werden Header hinzugefügt, die den Anforderungskontext bereitstellen.

HTTP-Methoden

Webfunktions-Endpunkte unterstützen`GET`,`HEAD`,`POST`,, `PUT` `PATCH`, `OPTIONS`, und. `DELETE`

Lambda lehnt eine nicht unterstützte Methode mit einer HTTP 400-Antwort ab, bevor die Anfrage Ihre Anwendung erreicht.

Anfordern von Headern

Lambda entfernt die folgenden Header, bevor es eine Anfrage weiterleitet:

- `Connection`, `Keep-Alive`, `Upgrade`, `Expect`, `TE`, und `Trailer`
- `Proxy-Authenticate`, `Proxy-Authorization` und `Proxy-Connection`
- `X-Real-IP`
- `X-HTTP-Method-Override`, `X-HTTP-Method` und `X-Method-Override`
- Header, deren Namen mit beginnen, und Header`lambda-web-`, die im Header benannt sind `Connection`

Lambda validiert oder ändert die folgenden Header.

Header	Behavior
`Host`	Muss der Domainname des Webfunktions-Endpunkts (z. B.`bcdfghjkl mnpqrst.b7t.lambda-web.us-east-1.on.aws`) in Kleinbuch staben und ohne Port sein. Lambda verwendet diesen Wert und gibt HTTP 404 zurück, wenn er keinem Endpunkt entspricht, oder HTTP 400, wenn der Header fehlt. Ihre Anwendung erhält den Domainnamen des Endpunkts.
`X-Forwarded-For`	Lambda fügt die IP-Adresse des verbindenden Clients als letzten Wert an. Frühere Werte stammen vom Client und wurden nicht validiert. Wenn sich eine CloudFront Amazon-Distribution vor dem Endpunkt befindet, ist der letzte Wert eine CloudFront Adresse.

Lambda leitet `X-Forwarded-Host` vom Client ohne Überprüfung weiter. Verwenden Sie es nicht, um Weiterleitungen, absolute URLs oder hostbasierte Autorisierungen zu erstellen, es sei denn, eine Eingangstür, die Sie kontrollieren, legt dies fest.

Lambda fügt die folgenden Header hinzu.

Header	Description
`lambda-web-request-id`	Der Bezeichner, den Lambda der Anfrage zuweist.
`lambda-web-endpoint-name`	Der Name des Endpunkts, der die Anfrage erhalten hat.

Bei Endpunkten, die die IAM-Autorisierung verwenden, validiert Lambda die signierte Anfrage und entfernt den `Authorization` Header, bevor sie weitergeleitet wird. Weitere Informationen finden Sie unter the section called “Verwenden Sie den `IamAuth`Authentifizierungstyp”.

Antwort-Header

Lambda entfernt verbindungsbezogene Header und HTTP-Trailer aus der Antwort Ihrer Anwendung. Abhängig vom Ergebnis der Anfrage kann Lambda die folgenden Header hinzufügen.

Header	Description
`lambda-web-request-id`	Die Anforderungs-ID. Verwenden Sie ihn, um eine Antwort mit Anwendungs- und Systemprotokollen zu korrelieren.
`lambda-web-logstream`	Der Amazon CloudWatch Logs-Protokollstream, der Protokolle für die Ausführungsumgebung enthält, die die Anfrage bearbeitet hat.
`lambda-web-error-type`	Der Fehlertyp im Format `<Prefix>.<ErrorReason>` (z. B. `Platform.EndpointRequestThrottle` oder`Runtime.RequestTimeout`). Lambda lässt diesen Header bei internen Dienstfehlern und Autorisierungsfehlern aus. Eine Liste der Werte finden Sie unter. the section called “Fehlertypen der Webfunktion”

Header-Grenzwerte

Die Anforderungszeile, die Anforderungsheader und die abschließende Leerzeile können insgesamt bis zu 128 KiB betragen. Lambda lehnt eine Anfrage ab, die dieses Limit mit HTTP 431 (Request Header Fields Too Large) überschreitet, bevor die Anfrage Ihre Anwendung erreicht. Informationen zur Fehlerbehebung finden Sie unter. the section called “Grenzwerte für Header anfordern”

Konfiguration von Netzwerken für AWS Lambda Web-Funktionen

Eine Webfunktion akzeptiert Anfragen von Clients. Es kann auch auf andere Ressourcen zugreifen, um seine Arbeit zu erledigen, z. B. eine Datenbank in Ihrer VPC, eine interne API oder einen Dienst im öffentlichen Internet. Sie konfigurieren diese beiden Richtungen unabhängig voneinander. Sie können einen Endpunkt für Anrufer verfügbar machen, ohne ihm VPC-Zugriff zu gewähren, oder einer Funktion privaten ausgehenden Zugriff gewähren, ohne zu ändern, wer sie aufrufen kann.

Eingehender Datenverkehr erreicht eine Webfunktion über einen Endpunkt. Der Endpunkt definiert seinen Domainnamen, seinen Endpunkttyp, seine Regionen und seine Autorisierungskonfiguration. Ausgehender Datenverkehr ist in einer Revision konfiguriert. Eine Version ohne ausgehenden Netzwerkconnector erreicht öffentliche `IPv4` Ziele direkt. Eine Revision, die auf einen Connector verweist, sendet seinen ausgehenden Datenverkehr über Ihre VPC. Eine Webfunktion läuft auf der Infrastruktur, die Lambda besitzt und verwaltet, nicht in Ihrer VPC. Ein ausgehender Netzwerkconnector ermöglicht nur ausgehenden VPC-Zugriff. Es ändert nichts daran, wie Clients den Endpunkt erreichen oder wer ihn aufrufen darf.

> **Note**
>
> Die Domainnamen der Webfunktions-Endpunkte sind über das öffentliche Internet erreichbar. AWS PrivateLink Schnittstellen-Endpunkte sind für Webfunktions-Endpunkt-Domainnamen nicht verfügbar. Verwenden Sie die Autorisierungskonfiguration des Endpunkts, um den Zugriff zu kontrollieren.

Eingehende Konnektivität

Clients rufen eine Webfunktion über einen Lambda-managed öffentlichen HTTPS-Endpunkt auf. Lambda beendet TLS und leitet die Anfrage an den HTTP-Server Ihrer Anwendung weiter. Der verwaltete Endpunkt unterstützt. HTTP/1.1 Es macht keine private VPC-Adresse verfügbar und akzeptiert keine direkten Verbindungen zur Ausführungsumgebung.

`IamAuth`steuert, wer einen Endpunkt aufrufen kann, macht den Endpunkt auf Netzwerkebene jedoch nicht privat. Informationen zu Autorisierungsoptionen finden Sie unterthe section called "Zugriffskontrolle".

Der Endpunkt steuert das Routing, die Autorisierung, die Drosselung und die Skalierung von Anfragen. Der Endpunkttyp und die eingestellte Region sind bei der Erstellung festgelegt. Um eine der Eigenschaften zu ändern, erstellen Sie einen neuen Endpunkt und verschieben Sie die Clients auf den neuen Domänennamen.

Die Endpunktkonfiguration ist unabhängig vom VPC-Ausgang. Das Hinzufügen eines ausgehenden Netzwerkconnectors zu einer Version macht den Endpunkt nicht privat und schränkt eingehende Anfragen nicht ein.

> **Important**
>
> Sie können ausgehende Netzwerkconnectors nur mit Endpunkten verwenden. `HomeRegion` Lambda lehnt Versuche ab, einen `MultiRegion` `PerRegion` OR-Endpunkt an eine Revision weiterzuleiten, die auf einen ausgehenden Netzwerkconnector verweist.

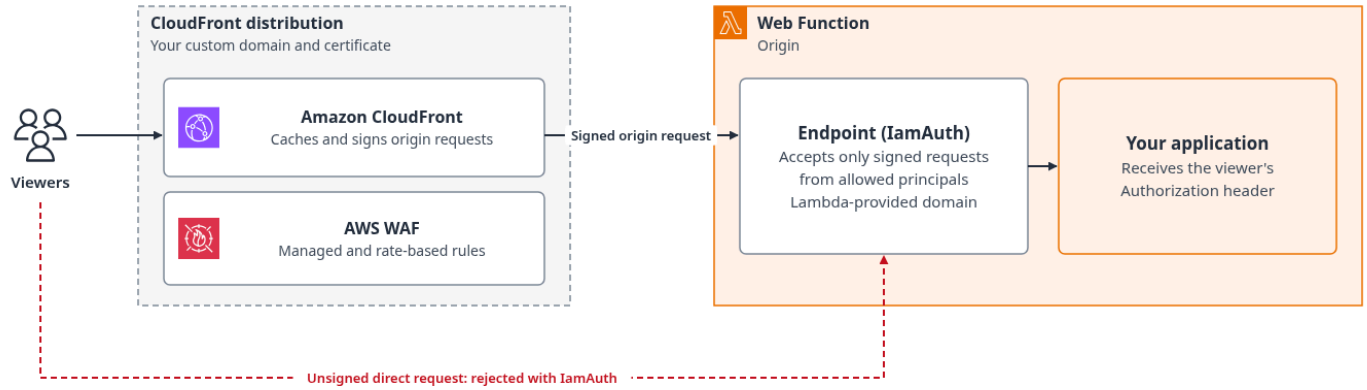
Wenn Revisionsgewichte den Datenverkehr zwischen Versionen mit unterschiedlichen Konnektorkonfigurationen aufteilen, verwenden Anfragen unterschiedliche Netzwerkpfade. Wenn Sie eine intermittierende Konnektivität diagnostizieren, überprüfen Sie zuerst den automatischen Bereitstellungsmodus und die Revisionsgewichte des Endpunkts. Informationen darüber, wie Anfragen an Revisionen weitergeleitet werden, finden Sie unter. the section called "Bereitstellungsmuster"

Fügen Sie eine HTTP-Vordertür vor einem Endpunkt hinzu

Eine Vordertür ist ein Dienst, der Kundenanfragen entgegennimmt und sie an Ihren Endpunkt weiterleitet. Er fügt Funktionen wie eine benutzerdefinierte Domäne, Caching oder Firewall-Schutz für Webanwendungen hinzu. Webfunktions-Endpunkte unterstützen keine benutzerdefinierten Domainnamen oder direkte AWS WAF Verknüpfungen. Verwenden Sie Amazon CloudFront vor dem Endpunkt, wenn Sie eine benutzerdefinierte Domain, Edge-Caching HTTP/2 oder HTTP/3 zwischen dem Viewer und dem Edge, Antwortkomprimierung oder AWS WAF Kontrollen benötigen.

Das folgende Diagramm zeigt eine Produktionstür. Ein Client stellt eine Verbindung zu einer CloudFront benutzerdefinierten Domäne her, die durch geschützt ist AWS WAF. CloudFront sendet

die Anfrage an einen geschützten Webfunktions-Endpunkt, der sie an eine Ausführungsumgebung weiterleitet. Ohne einen Ausgangsnetzwerkanschluss kann die Anwendung öffentliche AWS Dienste oder Internet-Endpunkte aufrufen. Es kann keine Ressourcen erreichen, die nur private VPC-Adressen haben. Informationen zum Erreichen dieser Ressourcen finden Sie unterthe section called “Eine Webfunktion mit Ihrer VPC verbinden”. Mit `IamAuth` CloudFront signiert jede ursprüngliche Anfrage, und Lambda autorisiert die Anfrage, bevor Ihre Funktion aufgerufen wird. Lambda behält den ursprünglichen `Authorization` Header des Viewers bei und leitet ihn an Ihre Funktion weiter. Die Lambda-provided Domain akzeptiert nur signierte Anfragen von Principals, die berechtigt sind, sie aufzurufen, und lehnt unsignierte direkte Anfragen ab, z. B. wenn ein Browser die Domain direkt aufruft. Um sicherzustellen, dass der Endpunkt nur Anrufe von Ihrer Distribution akzeptiert, fügen Sie eine ressourcenbasierte Richtlinie hinzu, die dem CloudFront Service Principal die Aufrufberechtigung erteilt und sie auf den Amazon-Ressourcennamen (ARN) Ihrer Distribution mit einer Quell-ARN-Bedingung beschränkt. Mit `ApplicationManaged` ist der Endpunkt öffentlich und direkte Anfragen umgehen und. CloudFront AWS WAF



1. Konfigurieren Sie die Web Function-Domain als benutzerdefinierten HTTPS-Ursprung.
2. Hängen Sie das Zertifikat und den alternativen Domainnamen an die Distribution an und verweisen Sie Ihre DNS-Einträge (Domain Name System) auf die Distribution.
3. Leiten Sie nur die Header, Cookies und Abfragezeichenfolgen weiter, die die Anwendung benötigt. Leiten Sie den `Host` Header des Viewers nicht weiter, da Lambda HTTP 404 weiterleitet `Host` und zurückgibt, wenn es sich nicht um die Endpunktdomäne handelt. Wenn Sie eine verwaltete Origin-Anforderungsrichtlinie verwenden, die alle Viewer-Header weiterleitet, wählen Sie. `AllViewerExceptHostHeader` Die Grenzwerte für Endpunkt-Header beinhalten weitergeleitete Header.
4. (Optional) Fügen Sie eine AWS WAF Web-ACL hinzu, wenn Sie eine IP-, Bot-, ratenbasierte oder verwaltete Regelprüfung benötigen.

5. Schützen Sie den Ursprung, sodass Clients die Verteilung nicht umgehen können. Gewähren Sie einem IAM-authorized Endpunkt nur Zugriff auf die vorgesehene Front-Door-Identität. Für eine anwendungsverwaltete Autorisierung benötigen und validieren Sie einen geheimen Ursprungsnachweis oder einen anderen Nachweis auf Anwendungsebene, dass die Anfrage über die Distribution eingegangen ist.

Informationen zum CloudFront Autorisierungsverfahren finden Sie unter. the section called “CloudFront Integration”

Browserzugriff und CORS

Lambda fügt keine Cross-Origin Resource Sharing (CORS) -Header für Webfunktionen hinzu. Ihre Anwendung oder Ihr Front-Door muss Preflight- und tatsächliche Anfragen bearbeiten.

- Gibt den zulässigen Ursprung zurück, anstatt jeden `Origin` Wert wiederzugeben.
- Erlaubt nur die Methoden und Anforderungsheader, die die Anwendung verwendet.
- Wenn Browser Anmeldeinformationen senden, verwenden Sie einen expliziten Ursprung und geben Sie ihn zurück`Access-Control-Allow-Credentials: true`. Kombinieren Sie Anmeldeinformationen nicht mit einem Platzhalter-Ursprung.
- Testen Sie `OPTIONS` Preflight-Anfragen getrennt von der Anwendungsmethode.
- Authentifizieren und autorisieren Sie jede geschützte Anfrage. CORS ist keine Authentifizierungskontrolle.

Streaming- und Protokolleinschränkungen

Webfunktionen streamen Anfragetexte an Ihre Anwendung, sobald sie eintreffen, und streamen Antworten zurück an den Client HTTP/1.1. Der Anforderungspfad ist Halbduplex: Lambda leitet den gesamten Anfragetext weiter, bevor es die Antwort Ihrer Anwendung liefert. Lesen Sie den Anfragetext vollständig, bevor Sie die Antwort schreiben. Stellen `timeoutSeconds` Sie ein, dass es das Streaming in beide Richtungen abdeckt: Empfangen des Anfragetextes und Senden der Antwort. Wenn das Timeout abgelaufen ist, beendet Lambda die Antwort. Der Client erhält den Statuscode, der bereits gesendet wurde, mit einem gekürzten Hauptteil. Beenden Sie die Upstream-Arbeit, wenn der Client die Verbindung trennt.

Lambda unterstützt keine WebSocket Upgrades. Lambda entfernt verbindungsspezifische Upgrade-Header, bevor die Anfrage weitergeleitet wird. Verwenden Sie vom Server gesendete Ereignisse, wenn ein unidirektionales Server-zu-Client-Streaming ausreichend ist.

Den vollständigen Header- und Methodenvertrag finden Sie unter. the section called “HTTP-Verhalten”

Ausgehende Konnektivität

Sie wählen den ausgehenden Netzwerkpfad für jede Revision.

- Ohne Ausgangsnetzwerkanschluss — Die Revision erreicht öffentliche `IPv4` Ziele direkt, ohne Ihre VPC zu durchqueren. Die Quelladresse ist Lambda-managed, ist nicht stabil und eignet sich nicht für eine Zulassungsliste. Die Revision hat keine Route zu privaten Ressourcen in Ihrer VPC.
- Mit einem ausgehenden Netzwerkconnector — Die Revision sendet ausgehenden Datenverkehr über elastische Netzwerkschnittstellen in den Subnetzen des Connectors. Es gelten Ihre VPC-Routing-Tabellen, Sicherheitsgruppen und Network Access Control Lists (ACLs). Verwenden Sie diese Option, um private VPC-Ressourcen, lokale Netzwerke, die über Direct Connect oder VPN verbunden sind, oder Ziele zu erreichen, die Datenverkehr aus Ihrem Adressraum benötigen.

Das Verweisen auf einen Connector ersetzt den standardmäßigen ausgehenden Pfad. Ein Connector leitet Traffic in Ihre VPC, bietet aber keinen Internetzugang. Um öffentliche `IPv4` Ziele zu erreichen, benötigen die Connector-Subnetze eine Route zu einem NAT-Gateway. Um öffentliche `IPv6` Ziele zu erreichen, benötigen sie eine Route zu einem Internet-Gateway, das nur für ausgehende Verbindungen bestimmt ist.

Es gibt keine stabile Quell-IP-Adresse für den Connector-Verkehr. Konfigurieren Sie die Zielsicherheitsgruppen so, dass sie die Sicherheitsgruppen des Connectors zulassen. Wenn ein Ziel keine Sicherheitsgruppenverweise unterstützt, lassen Sie die CIDR-Blöcke des Connector-Subnetzes anstelle einer einzelnen Adresse zu.

Lambda führt Connector-Operationen asynchron aus. Ein Konnektor meldet, `PENDING` während Lambda seine Netzwerkschnittstellen erstellt und `ACTIVE` wann Revisionen darauf verweisen können. Zu den anderen Staaten gehören`INACTIVE`, `FAILEDDELETING`, und. `DELETE_FAILED`

Suchen Sie nach einem Update `LastUpdateStatus` nach `InProgressSuccessful`, oder`Failed`. Wenn ein Vorgang fehlschlägt, identifizieren Sie anhand des Zustandsgrundes und des Ursachencodes das betroffene Subnetz, die Sicherheitsgruppe, die betroffenen Berechtigungen oder die betroffene IP-Adresskapazität.

DNS-Auflösung

Aktivieren Sie für private DNS-Namen die DNS-Unterstützung und DNS-Hostnamen in der VPC des Connectors. Ordnen Sie dieser VPC alle erforderlichen privaten Hosting-Zonen von Route 53 zu.

In jeder Ausführungsumgebung werden DNS-Antworten zwischengespeichert. Nach der Änderung eines DNS-Eintrags verwenden bestehende Umgebungen weiterhin die alte Antwort, bis der zwischengespeicherte Wert abläuft oder Lambda die Umgebung ersetzt. Um einen sauberen Rollout zu erzwingen, erstellen Sie eine neue Version und verlagern Sie den Endpunktdatenverkehr auf diese Version.

Testen Sie bei der Fehlerbehebung die Namensauflösung und die Erreichbarkeit über das Netzwerk getrennt. Stellen Sie über die IP-Adresse eine Connect zu einem Ziel her, um den Netzwerkpfad zu überprüfen, und testen Sie dann den DNS-Namen.

Eine Webfunktion mit Ihrer VPC verbinden

Ein ausgehender Netzwerkkonnektor ist eine wiederverwendbare Lambda-Ressource. Sie erstellen den Connector einmal und verweisen auf eine veröffentlichte Connector-Version aus jeder Webfunktions-Revision, die VPC-Zugriff benötigt.

Für Web Functions hat die Connector-Referenz die folgenden Eigenschaften:

- Eine Revision verweist auf keinen oder einen Konnektor.
- Der Connector-ARN muss einen Versionskennzeichner enthalten.
- Der Connector muss sich im selben AWS Konto und in derselben Region wie die Webfunktion befinden.
- Dem Connector muss ein `WebFunction` Computing-Ressourcentyp zugeordnet sein.
- Nur ein `HomeRegion` Endpunkt unterstützt Revisionen, die auf einen ausgehenden Netzwerkconnector verweisen.
- Sie können die Referenz nicht mehr ändern, nachdem Sie sie festgelegt haben. Um sie hinzuzufügen, zu ändern oder zu entfernen, erstellen Sie eine neue Revision.

Voraussetzungen

Bevor Sie den Konnektor erstellen:

- Wählen Sie mindestens zwei Subnetze in verschiedenen Availability Zones mit verfügbaren IP-Adressen aus.
- Wählen Sie Sicherheitsgruppen aus, die den erforderlichen ausgehenden Datenverkehr zulassen.
- Lassen Sie eingehenden Datenverkehr von den Sicherheitsgruppen des Connectors an jedem Ziel zu.
- Konfigurieren Sie Routen von den Connector-Subnetzen zu den erforderlichen privaten, lokalen oder öffentlichen Zielen.
- Erstellen Sie eine Operatorrolle, die Lambda die Berechtigung zur Verwaltung von Netzwerkschnittstellen gewährt.
- Stellen Sie sicher, dass die Identität, die den Konnektor und die Revision erstellt, über die erforderlichen Berechtigungen verfügtthe section called “Berechtigungen für die Netzwerkkonfiguration”, auch `lambda:PassNetworkConnector` für den Konnektor und `iam:PassRole` die Operatorrolle.

Die Operatorrolle des Connectors und die Ausführungsrolle der Revision dienen unterschiedlichen Zwecken. Die Operatorrolle verwaltet Netzwerkschnittstellen in Ihrer VPC. Die Ausführungsrolle erteilt Ihrer Anwendung die Erlaubnis, AWS APIs aufzurufen. Sie fügen der Ausführungsrolle keine Connector-Netzwerkschnittstellenberechtigungen hinzu.

Berechtigungen für die Operatorrolle

Lambda übernimmt die Betreiberrolle des Connectors zur Verwaltung der Netzwerkschnittstellen des Connectors. Die Vertrauensrichtlinie der Rolle muss es dem Lambda-Serviceprinzipal, ermöglichen`lambda.amazonaws.com`, sie zu übernehmen:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Fügen Sie der Operatorrolle des Connectors eine Berechtigungsrichtlinie wie die folgende hinzu. Diese Richtlinie ermöglicht es Lambda, die Netzwerkschnittstellen des Connectors zu erstellen und zu kennzeichnen:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CreateENI",
      "Effect": "Allow",
      "Action": "ec2:CreateNetworkInterface",
      "Resource": [
        "arn:aws:ec2:*:*:network-interface/*",
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Sid": "TagENI",
      "Effect": "Allow",
      "Action": "ec2:CreateTags",
      "Resource": "arn:aws:ec2:*:*:network-interface/*",
      "Condition": {
        "StringEquals": {
          "ec2:ManagedResourceOperator": "network-connectors.lambda.amazonaws.com"
        }
      }
    }
  ]
}
```

Erstellen eines Connectors von

Der folgende AWS CLI Befehl erstellt einen `IPv4` Connector für Webfunktionen. Ersetzen Sie den Connectornamen, die Subnetz-IDs, die Sicherheitsgruppen-ID und den ARN der Operatorrolle durch Ihre eigenen Werte.

```
aws lambda-core create-network-connector \
  --name my-connector \
  --configuration '{
    "VpcEgressConfiguration": {
      "SubnetIds": [
```

```
        "subnet-0ab1c2d3e4f5a6b78",
        "subnet-1bc2d3e4f5a6b7c89"
      ],
      "SecurityGroupIds": ["sg-0123456789abcdef0"],
      "NetworkProtocol": "IPv4",
      "AssociatedComputeResourceTypes": ["WebFunction"]
    }
  }' \
  --operator-role arn:aws:iam::123456789012:role/NetworkConnectorOperatorRole
```

Ein Connector unterstützt einen Computing-Ressourcentyp. Ein Connector für Web Functions unterstützt keinen anderen Lambda-Berechnungstyp, und sein Rechenressourcentyp ist unveränderlich.

Lambda erstellt den Konnektor asynchron. Verwenden Sie diese `get-network-connector` Option, um zu warten, bis dies `State` der Fall ist`ACTIVE`, und notieren Sie sich die aktuelle Versionsnummer.

Referenzieren Sie den Stecker aus einer Revision

`serviceConfig.egressNetworkConnectorArn`Wird festgelegt, wenn Sie die erste oder eine spätere Version der Web-Funktion erstellen. Verwenden Sie einen versionsqualifizierten ARN:

```
arn:aws:lambda:us-east-1:123456789012:network-connector:nc-0a1b2c3d4e5f6a7b8:4
```

Der folgende AWS CLI Befehl erstellt eine Revision, die auf Connector-Version 4 verweist. Ersetzen Sie den Funktionsnamen, den ARN der Ausführungsrolle, den Konnektor-ARN und die Region durch Ihre eigenen Werte.

```
aws lambda-web create-web-function-revision \
  --function-name my-web-function \
  --build-config file://build-config.json \
  --service-config '{
    "executionRoleArn": "arn:aws:iam::123456789012:role/WebFunctionExecutionRole",
    "timeoutSeconds": 30,
    "egressNetworkConnectorArn": "arn:aws:lambda:us-east-1:123456789012:network-
connector:nc-0a1b2c3d4e5f6a7b8:4"
  }' \
  --region us-east-1
```

Ein unqualifizierter ARN, Connectorname oder Connector-ID wird für eine Revision nicht akzeptiert. Nach dem Erstellen der Revision:

- Warten Sie, bis der Status der Revision lautet`Active`.
- Lesen Sie die Revision und vergewissern Sie sich, dass sie die erwartete Connector-Version `egressNetworkConnectorArn` enthält.
- Vergewissern Sie sich, dass der Endpunkt den Datenverkehr an die Revision weiterleitet. Ein `HomeRegion` Endpunkt im `LatestRevision` Modus nimmt die letzte aktive Revision automatisch auf. Aktualisieren Sie für einen Endpunkt im `Disabled` Modus seine Revisionsgewichte.
- Testen Sie ein VPC-Ziel und bestätigen Sie den Datenverkehr mit VPC Flow Logs.

Der automatische Bereitstellungsmodus des Endpunkts bestimmt, ob das Routing beim Erstellen einer aktiven Version automatisch aktualisiert wird. Vergewissern Sie sich in allen Modi, welche Version verfügbar ist, bevor Sie den zugehörigen Konnektor testen.

VPC-Konnektivität aktualisieren oder entfernen

Beim Aktualisieren eines Connectors wird eine neue Version veröffentlicht. Bestehende Revisionen bleiben an die Version in ihrem Connector-ARN ARN.

So führen Sie einen Konnektorwechsel durch:

- Aktualisieren Sie den Konnektor mit`update-network-connector`. Der Anschluss muss sein`ACTIVE`. Geben Sie die vollständige Konfiguration an`VpcEgressConfiguration`, einschließlich aller Subnetz-ID und Sicherheitsgruppen-ID, die Sie behalten möchten, da das Update die bestehende Konfiguration ersetzt.
- Warten Sie, `LastUpdateStatus` bis der Connector wird`Successful`.
- Holen Sie sich die neue Connector-Version.
- Erstellen Sie eine Revision, die auf den neuen versionsqualifizierten ARN verweist.
- Vergewissern Sie sich bei einem `HomeRegion` Endpunkt im `LatestRevision` Modus, dass er die neue aktive Revision übernommen hat. Bei einem Endpunkt im `Disabled` Modus verlagern Sie den Datenverkehr, indem Sie die Gewichtungen der Revision aktualisieren.

Wenn Sie den Connector aktualisieren, ohne eine Version zu erstellen, die auf die neue Version verweist, ändert sich nichts an der Netzwerkkonfiguration der laufenden Webfunktionen.

Um die VPC-Konnektivität zu entfernen, erstellen Sie eine Version ohne`egressNetworkConnectorArn`. Ein `HomeRegion` Endpunkt im `LatestRevision` Modus nimmt ihn auf, wenn er aktiv wird. Bei einem Endpunkt im `Disabled` Modus können Sie den Datenverkehr verlagern, indem Sie seine Revisionsgewichte aktualisieren. Die neue Version verwendet Lambda-managed öffentlichen `IPv4` Ausgang und hat keinen privaten VPC-Zugriff mehr.

> **Note**
>
> Bevor Sie einen ausgehenden Netzwerkconnector löschen, verlagern Sie den Endpunktdatenverkehr von allen Web Function-Revisionen, die auf eine beliebige Version des Connectors verweisen, und löschen Sie dann diese Revisionen. Wenn Sie einen Connector löschen, während eine aktive Version noch auf ihn verweist, wird die Netzwerkkonnektivität sowohl für bestehende als auch für neue Ausführungsumgebungen, die diese Revision bereitstellen, unterbrochen.

IPv6-Support

Nur der standardmäßige ausgehende Pfad und ein `IPv4` Connector werden unterstütztIPv4. Um ihn zu verwendenIPv6, konfigurieren Sie den Connector mit der `NetworkProtocol` Einstellung auf`DualStack`.

Für Dual-Stack-Konnektivität:

- Die VPC- und Connector-Subnetze müssen über `IPv6` CIDR-Blöcke verfügen.
- Routentabellen müssen Routen für die erforderlichen Ziele enthalten. `IPv6`
- Sicherheitsgruppen und Netzwerk-ACLs müssen `IPv6` Datenverkehr zulassen.
- Für `IPv6` den öffentlichen Zugriff ist eine: :/0-Route zu einem Internet-Gateway nur für ausgehenden Datenverkehr erforderlich.
- Verweisen Sie auf die Connector-Version, die erstellt wurde, als Sie den Connector auf aktualisiert haben. `DualStack`

Ermitteln Sie die `IPv6` Verfügbarkeit nicht, indem Sie Schnittstellen innerhalb einer Ausführungsumgebung überprüfen. Überprüfen Sie dies, indem Sie eine `IPv6` Verbindung über den Anwendungscode herstellen und die VPC Flow Logs überprüfen.

Skalierungsverhalten für Lambda Web Functions

Lambda skaliert den Endpunkt Ihrer Web Function automatisch, wenn der Traffic eintrifft. Es skaliert in zwei Dimensionen:

- Anzahl der Umgebungen — Lambda fügt weitere Ausführungsumgebungen hinzu, bis zu `maxEnvironments`.
- Gleichzeitige Anfragen pro Umgebung — Jede Umgebung verarbeitet mehrere Anfragen gleichzeitig. Lambda verwaltet ein dynamisches Parallelitätslimit pro Umgebung, das auf Ihrer Ressourcennutzung basiert. Sie steuern die Obergrenze dieser dynamischen Grenze mit `maxConcurrencyPerEnvironment`. Weitere Informationen finden Sie unter the section called “Dynamisches Parallelitätslimit pro Umgebung”.

Sie können auch einstellen `rateLimit` auf einem Endpunkt, um die Empfangsrate seiner Anfragen zu begrenzen. Dies ist eine optionale Einstellung und wird angewendet, bevor Anfragen Ihre Umgebungen erreichen.

Ein Endpunkt skaliert von Null aus. Wenn kein Verkehr stattfindet, führt Lambda keine Umgebungen aus. Auf dieser Seite werden die Skalierungseinstellungen eines Endpunkts, die zugehörigen Kontokontingente und die Auswahl von Werten zur Kosten- oder Leistungsoptimierung erläutert.

Ausführungsumgebungen

Eine Ausführungsumgebung ist eine isolierte Recheninstanz, auf der Ihr HTTP-Server ausgeführt wird. Jede Umgebung hat ihren eigenen Speicher, ihr eigenes Dateisystem und ihren eigenen Prozessraum. Informationen dazu, wie Lambda eine Ausführungsumgebung initialisiert, wiederverwendet und herunterfährt, finden Sie unter the section called “Lebenszyklus der Umgebung”.

Lambda erstellt und entfernt automatisch Umgebungen auf der Grundlage des Datenverkehrs:

- Wenn der Verkehr zunimmt, erstellt Lambda zusätzliche Umgebungen, um die Verkehrsnachfrage zu decken.
- Wenn der Verkehr abnimmt, leert Lambda nicht ausgelastete Umgebungen und entfernt sie.
- Wenn kein Verkehr stattfindet, entfernt Lambda alle Umgebungen. Der Endpunkt skaliert auf Null.

Es gibt keine Mindesteinstellung oder Einstellung für eine reservierte Umgebung. Ein Endpunkt wird immer von Null aus skaliert, wenn der Verkehr eintrifft. Um Kaltstarts bei geringer Zunahme des Datenverkehrs zu vermeiden, skaliert Lambda die Ausführungsumgebungen automatisch vor Bedarf, um freie Kapazitäten (Headroom) für eingehenden Verkehr aufrechtzuerhalten. Wenn Umgebungen nicht ausreichend ausgelastet sind, skaliert Lambda sie herunter, auch auf Null.

Dynamisches Parallelitätslimit pro Umgebung

Das dynamische Parallelitätslimit pro Umgebung ist ein Lambda-managed Begrenzung der Anzahl von Anforderungen, die eine einzelne Ausführungsumgebung gleichzeitig verarbeitet. Basierend auf der Ressourcenauslastung Ihrer Funktion passt Lambda dieses Limit automatisch an, um Effizienz und Leistung zu optimieren. Lambda skaliert neue Umgebungen, wenn bestehende Umgebungen sich ihrem Limit nähern. Wenn das Limit verhindert, dass alle vorhandenen Umgebungen mehr Anfragen annehmen, stellen sich Anfragen in eine Warteschlange und warten auf die nächste verfügbare Umgebung.

Für eine neue Webfunktion legt Lambda ein konservatives Startlimit fest und passt es schnell an die Ressourcennutzung der Funktion an. Weitere Informationen darüber, wie sich das Limit während der Lastverarbeitung ändert, finden Sie unter the section called “Dynamisches Limit für die Parallelität einer neuen Webfunktion”.

Sie konfigurieren das dynamische Limit nicht direkt. Stattdessen können Sie eine Obergrenze darauf anwenden, indem Sie Folgendes festlegen `maxConcurrencyPerEnvironment` zur Revision. Die dynamische Grenze von Lambda kann unter diesem Wert liegen, überschreitet ihn jedoch niemals. Zum Beispiel, wenn Sie Folgendes festlegen `maxConcurrencyPerEnvironment` Auf 32 leitet Lambda maximal 32 gleichzeitige Anfragen an eine beliebige Umgebung weiter, auch wenn die Umgebung über Kapazitäten für mehr verfügt.

Die meisten Anwendungen können sich auf das dynamische Limit verlassen, ohne es zu ändern `maxConcurrencyPerEnvironment`. Hinweise, wann Sie es ändern sollten, finden Sie unter the section called “Auswahl der Skalierungseinstellungen”.

Einstellungen skalieren

Drei Einstellungen steuern, wie Lambda Ihre Web Function skaliert. Sie konfigurieren jede Einstellung entweder auf dem Endpunkt oder in der Revision.

Einstellung	Konfiguriert am	Gültiger Bereich	Standard	Description
`maxConcurrencyPerEnvironment`	Revision	1—128	64	Die obere Grenze des Lambda-managed dynamischen Parallelitätslimit pro Umgebung. Lambda passt die tatsächliche Parallelität pro Umgebung automatisch an, sodass sich die meisten Anwendungen auf das dynamische Limit verlassen können, ohne diesen Wert zu ändern.
`maxEnvironments`	Endpoint	Optional. 2—10.000	Keine	Schränkt die Anzahl der Ausführungsumgebungen für den jeweiligen Endpunkt ein AWS-Region. Steuert die horizontale Skala. Die Skalierung der Endpunktumgebung ist immer durch das verfügbare vCPU-Kontingent des Kontos begrenzt, auch wenn `maxEnvironments` ist nicht festgelegt.
`rateLimit`	Endpoint	Optional. 0—10.000	Keine	Begrenzt die Anforderungsrate des Endpunkts, ausgedrückt in Anfragen pro Sekunde, in jeder AWS-Region. Werte bis 1.000 müssen ein Vielfaches von 100 sein, und Werte über 1.000 müssen ein Vielfaches von 1.000 sein. Ein Wert von 0 blockiert den gesamten Verkehr. Anfragen,

Einstellung	Konfiguriert am	Gültiger Bereich	Standard	Description
				die über dem effektiven Ratenlimit liegen, erhalten eine HTTP-429-Antwort (Too Many Requests). Wenn diese Option nicht festgelegt ist, wird die Empfangsrate des Endpunkts durch das Kontingent pro Endpunkt pro Konto begrenzt.

`maxEnvironments` und `rateLimit` sind optionale Endpunkteigenschaften. Sie können sie aktualisieren, ohne eine neue Revision zu erstellen. Standardmäßig ist keines von beiden festgelegt. Ein großes `maxEnvironments` garantiert nicht, dass der Endpunkt so weit skaliert werden kann. Die Skalierung der Umgebung ist immer durch das regionale vCPU-Kontingent des Kontos begrenzt. EIN `rateLimit` von 0 blockiert den gesamten Verkehr. Details hierzu finden Sie unter the section called “Drosselung”. Die Anforderungsrate am Endpunkt ist immer durch das Kontingent für die Anforderungsrate des Kontos begrenzt, auch wenn `rateLimit` ist nicht festgelegt. Die Kontingente finden Sie im nächsten Abschnitt.

Um zu ändern `maxEnvironments` oder `rateLimit`, lauf `aws lambda-web update-web-function-endpoint` mit dem `--scaling-config` oder `--throttle-config` Parameter, zum Beispiel `--scaling-config '{"maxEnvironments": 50}'` oder `--throttle-config '{"rateLimit": 8000}'`. Lassen Sie einen Parameter weg, um seinen aktuellen Wert beizubehalten. Um einen von Ihnen festgelegten Wert zu löschen, übergeben Sie ein leeres Objekt, z. B. `--scaling-config '{}'`. Weitere Informationen finden Sie unter the section called “Aktualisierung eines Endpunkts”. `maxConcurrencyPerEnvironment` ist eine Revisionseinstellung. Um sie zu ändern, erstellen Sie eine neue Revision. Ob die neue Revision Datenverkehr empfängt, hängt vom automatischen Bereitstellungsmodus des Endpunkts ab. Weitere Informationen finden Sie unter the section called “Eine Revision erstellen”.

Kontingente für Konten

Diese Kontingente gelten pro AWS-Region. Sie gelten immer zusätzlich zu Ihren optional konfigurierten Endpunktlimits.

- Per-endpoint Ratenlimit — 10.000 Anfragen pro Sekunde (RPS).
- Gesamtratenlimit für Konten — 100.000 RPS für alle Endpunkte in der Region.
- vCPU-Limit des Kontos — 2.000 vCPUs auf allen Endpunkten in der Region. Jede Ausführungsumgebung verbraucht 2 vCPUs, daher begrenzt dieses Kontingent die Gesamtzahl der Umgebungen pro AWS-Region auf dem gesamten Konto.

Die aktuellen Kontingentwerte und wie Sie eine Erhöhung beantragen können, finden Sie unter the section called “Lambda-Web-Funktionen”.

Wie die Skalierungseinstellungen interagieren

Diese drei Skalierungseinstellungen wirken unabhängig voneinander, um die Skalierung Ihres Endpunkts zu bestimmen. Mit Ausnahme von Kontokontingenten skaliert jeder Endpunkt unabhängig, auch Endpunkte, die zu derselben Webfunktion gehören.

Gesamtkapazität für Parallelität

Sie können sich die maximale Anzahl der Anfragen während des Fluges am Endpunkt als ungefähr vorstellen `maxEnvironments` multipliziert mit dem dynamischen Parallelitätslimit pro Umgebung, das Lambda derzeit zulässt. Das Höchste, was der Endpunkt gleichzeitig verarbeiten kann, ist `maxEnvironments` × `maxConcurrencyPerEnvironment`. Es erreicht diese Zahl nur, wenn es läuft `maxEnvironments` Umgebungen und jede einzelne behandelt `maxConcurrencyPerEnvironment` Anfragen. Wenn `maxEnvironments` nicht festgelegt ist, skaliert der Endpunkt je nach Bedarf bis zum vCPU-Kontingent des Kontos.

Obergrenze für die Gesamtrate der Anfragen

`rateLimit` begrenzt Anfragen pro Sekunde, unabhängig von der Parallelitätskapazität. Lambda gibt HTTP 429 für Anfragen zurück, die das Ratenlimit überschreiten, auch wenn die Umgebungen über Kapazitäten verfügen, um sie zu verarbeiten.

Hochskalieren

Bevor alle Umgebungen das dynamische Parallelitätslimit erreichen, skaliert Lambda weitere Umgebungen, bis zu `maxEnvironments`. Ein Endpunkt wird nicht mehr hochskaliert, wenn er `maxEnvironments` oder das vCPU-Kontingent des Kontos. Wenn ein Endpunkt nicht hochskaliert werden kann und alle vorhandenen Umgebungen voll ausgelastet sind, drosselt Lambda neue Anfragen sofort mit einer HTTP-429-Antwort (Too Many Requests). Lambda stellt gedrosselte Anfragen nicht in die Warteschlange. Wenn eine neue Ausführungsumgebung

initialisiert wird, werden Anfragen in die Warteschlange gestellt, anstatt sofort gedrosselt zu werden.

Vorübergehendes Überschwingen

Bei Gewichtsänderungen, internen Flottenzyklen und anderen Betriebsereignissen kann Lambda vorübergehend einen Wert überschreiten `maxEnvironments` um der Verfügbarkeit Priorität einzuräumen. Diese Überschreitung ist kurz und Lambda kehrt automatisch zum konfigurierten Limit zurück. Sie müssen keinen Spielraum für diese Betriebsereignisse einplanen, wenn Sie dies wünschen `maxEnvironments`.

Example Beispiel: Kapazitätsberechnung

Sie konfigurieren die folgenden Einstellungen für Ihre Web-Funktion:

- `maxEnvironments` = 10
- `maxConcurrencyPerEnvironment` = 64
- `rateLimit` = 8000

Diese Einstellungen begrenzen die Parallelität (Anfragen gleichzeitig während des Fluges) und die Anforderungsrate (Anfragen werden pro Sekunde akzeptiert) getrennt. Mit bis zu 10 Umgebungen und jeweils bis zu 64 gleichzeitigen Anfragen kann der Endpunkt maximal etwa 640 gleichzeitige Anfragen (10 × 64) bearbeiten. Das dynamische Limit ist oft niedriger, sodass die anhaltende Parallelität in der Regel geringer ist. Welches Limit der Endpunkt zuerst erreicht, hängt davon ab, wie lange jede Anfrage dauert.

Concurrency-bound Endpunkt

Nehmen wir an, dass jede Anfrage 200 ms dauert, zum Beispiel weil sie auf eine Datenbankabfrage wartet. Bei 640 laufenden Anfragen erledigt der Endpunkt höchstens etwa 3.200 Anfragen pro Sekunde (640 ÷ 0,2 s), was unter 8.000 RPS liegt `rateLimit`. Wenn alle 10 Umgebungen voll ausgelastet sind, drosselt Lambda neue Anfragen mit HTTP 429. Um mehr Traffic abzuwickeln, erhöhen Sie `maxEnvironments`.

Request-rate-bound Endpunkt

Angenommen, jede Anfrage dauert 10 ms. Bei 8.000 Anfragen pro Sekunde sind etwa 80 Anfragen im Gange (8.000 × 0,01 s), deutlich unter 640. Lambda gibt HTTP 429 für Anfragen über 8.000 RPS zurück, obwohl die Umgebungen über Kapazitäten verfügen, um sie zu verarbeiten.

Um mehr Anfragen anzunehmen, erhöhen Sie `rateLimit`, bis zum Kontingent von 10.000 RPS pro Endpunkt.

Drosselung

Wenn die Rate eingehender Anfragen die überschreitet `rateLimit` Bei Einstellung dieser Einstellung gibt Lambda bei überzähligen Anfragen eine HTTP 429-Antwort (Too Many Requests) zurück. Die Ratenbegrenzung gilt jeweils für den Endpunkt als Ganzes AWS-Region.

Lambda gibt auch eine HTTP 429-Antwort zurück, wenn alle Umgebungen voll ausgelastet sind, entweder weil der Endpunkt erreicht ist `maxEnvironments` oder das Konto hat sein vCPU-Kontingent erreicht. Weitere Informationen finden Sie unter the section called “Wie die Skalierungseinstellungen interagieren”.

Lambda kann HTTP 429 auch mit dem zurückgeben `Platform.EndpointConcurrentInitThrottle` Fehlertyp, wenn zu viele Ausführungsumgebungen gleichzeitig initialisiert werden. Diese Drosselklappe erhöht keine Drosselkennzahl. Die Metriken, die die anderen Drosselarten erhöhen, finden Sie unter the section called “Metriken”.

Wenn ein Endpunkt `rateLimit` ist 0, gibt Lambda HTTP 503 (Service Unavailable) statt HTTP 429 zurück, mit `lambda-web-error-type` Kopfzeile `Platform.EndpointDisabled`. Lambda kann auch HTTP 503 zurückgeben, wenn es Last abgibt, um den Dienst zu schützen. Versuchen Sie es erneut mit 503, der keinen hat `lambda-web-error-type` Header mit Backoff und Jitter. Ein 503 mit `Platform.EndpointDisabled` fährt fort, bis Sie eingestellt sind `rateLimit` auf einen Wert ungleich Null.

Lambda misst das Ratenlimit pro Sekunde. Kurzfristige Überschreitungen des Grenzwerts könnten akzeptiert werden. Dauernder Verkehr über dem Grenzwert führt zu Drosselung.

Wenn Ihre Anwendung HTTP-429-Antworten empfängt, implementieren Sie exponentielles Backoff mit Jitter in Ihrem Client-Code, um gedrosselte Anfragen erneut zu versuchen.

Hochskalieren und Herunterskalieren

Aufwärtsskalierung

Wenn der Datenverkehr zunimmt und sich bestehende Umgebungen dem dynamischen Grenzwert für Parallelität pro Umgebung nähern, erstellt Lambda zusätzliche Ausführungsumgebungen. Jede

neue Umgebung wird initialisiert, bevor sie Anforderungen verarbeitet. Die Schritte zur Initialisierung und das Zeitlimit für den Start finden Sie unter the section called “Initialisierung (Kaltstart)”.

Während eine neue Umgebung initialisiert wird, leitet Lambda Anfragen an bestehende Umgebungen weiter, die über verfügbare Kapazitäten verfügen. Wenn keine Umgebung über Kapazitäten verfügt, wartet die Anfrage, bis Lambda eine neue initialisiert (Kaltstart). Lambda gibt HTTP 429 nur zurück, wenn einer der Grenzwerte the section called “Drosselung” gilt, wie `maxEnvironments` oder das vCPU-Kontingent des Kontos.

Verkleinern

Wenn der Datenverkehr abnimmt, identifiziert Lambda nicht ausgelastete Umgebungen und entfernt sie. Lambda entleert eine Umgebung, bevor sie entfernt wird. Es stoppt die Weiterleitung neuer Anfragen an die Umgebung, lässt Anfragen während des Fluges nach bestem Bemühen abschließen und sendet `SIGTERM`. Details hierzu finden Sie unter the section called “Lebenszyklus der Umgebung”.

Lambda wird auf null Umgebungen herunterskaliert, wenn kein Verkehr stattfindet. Die nächste Anfrage danach startet eine neue Umgebung (ein Kaltstart).

Auswahl der Skalierungseinstellungen

Beachten Sie Folgendes, wenn Sie Werte für den Endpunkt auswählen `rateLimit`, Endpunkt `maxEnvironments`, und Revision `maxConcurrencyPerEnvironment`.

Wählend `rateLimit`

Der Standardwert ist „Keine“, was bedeutet, dass die Anforderungsrate eines Endpunkts sowohl durch das RPS-Kontingent von 10.000 pro Endpunkt als auch durch das RPS-Kontingent von 100.000 Konten begrenzt ist. Erwägen Sie, eine festzulegen `rateLimit` Wert, wenn Sie einschränken möchten, wie viel ein Endpunkt von der RPS-Quote Ihres Kontos beanspruchen kann, die von allen Ihren Endpunkten gemeinsam genutzt wird.

Einstellung eines `rateLimit` kann auch hilfreich sein, um den gesamten Verkehr auf nachgelagerte Abhängigkeiten zu beschränken.

Wählen `maxEnvironments`

Der Standardwert ist „Keine“, was bedeutet, dass die Anzahl der Umgebungen eines Endpunkts nur durch das vCPU-Kontingent des Kontos begrenzt ist. Erwägen Sie die Einstellung eines

`maxEnvironments` Wert, wenn Sie einschränken möchten, wie viel ein Endpunkt vom vCPU-Kontingent Ihres Kontos beanspruchen kann, das von allen Ihren Endpunkten gemeinsam genutzt wird.

Beim Einstellen eines `maxEnvironments` Bei einem Wert sollten Sie erwägen, die Anzahl der Umgebungen des Endpunkts zu überwachen, damit Sie benachrichtigt werden, wenn er sich in der Nähe Ihres `maxEnvironments` begrenzen und Maßnahmen ergreifen, um eine Drosselung für eine Anwendung zu vermeiden, die im Laufe der Zeit organisch wächst.

Einstellung eines niedrigen Werts von `maxEnvironments` kann dazu beitragen, die Kosten für den Betrieb von Umgebungen für Ihren Endpunkt zu begrenzen. Erwägen Sie, für nicht kritische Hobbyprojekte einen niedrigen Wert zu verwenden.

Wählen `maxConcurrencyPerEnvironment`

Eine Revision `maxConcurrencyPerEnvironment` ist die Obergrenze für das von Lambda verwaltete dynamische Parallelitätslimit pro Umgebung. Der Standardwert ist 64, und die meisten Benutzer sollten keinen optimalen Wert auswählen müssen.

Für speichergebundene Anwendungen die Einstellung `maxConcurrencyPerEnvironment` Ein niedrigerer Wert als der Standardwert kann dazu beitragen, ein vorhersehbareres Verhalten zu erzielen.

Für CPU-bound Anwendungen, Einstellung `maxConcurrencyPerEnvironment` Ein niedrigerer Wert als der Standardwert kann die CPU-Auslastung verringern und die Steady-State-Latenz in warmen Umgebungen verbessern, kann aber auch zu häufigeren Kaltstarts führen. Einstellung `maxConcurrencyPerEnvironment` Ein Wert, der höher als der Standardwert ist, hat wahrscheinlich keine Auswirkung, wenn das von Lambda verwaltete dynamische Parallelitätslimit den Datenverkehr in der Umgebung einschränkt.

Für I/O-bound Anwendungen, Einstellung `maxConcurrencyPerEnvironment` Eine höhere als die Standardeinstellung kann die Auslastung pro Umgebung verbessern und die Kosten senken.

Überwachen Sie Folgendes CloudWatch Metriken, um festzustellen, ob Ihre Skalierungseinstellungen angemessen sind. Definitionen von Metriken finden Sie unter the section called “Metriken”.

- `EndpointRequestThrottles` und `AccountRequestThrottles` — Wenn `EndpointRequestThrottles` konstant größer als Null ist, erhöhen `rateLimit`, bis zum Kontingent von 10.000 RPS pro Endpunkt. `AccountRequestThrottles` wird erhöht, wenn das

Konto sein Kontingent für die Anforderungsrate in der Region erreicht. Um diese Drosselungen zu verringern, fordern Sie eine Erhöhung oder Senkung der Quote an `rateLimit` auf anderen Endpunkten.

- `EndpointActiveEnvironmentThrottles` und `AccountActiveVCPUArm64Throttles` — Diese Metriken erhöhen sich, wenn `maxEnvironments` oder das vCPU-Kontingent des Kontos verhindert zusätzliches Scale-Out. Um diese Drosselungen zu verringern, erhöhen Sie `maxEnvironments` oder fordern Sie eine Erhöhung der vCPU-Quote an.
- `EndpointActiveEnvironments` und `EnvironmentConcurrentRequests` — Wenn sich aktive Umgebungen nähern `maxEnvironments`, oder gleichzeitige Anfragen pro Umgebung bleiben in der Nähe `maxConcurrencyPerEnvironment`, erhöhen Sie die entsprechende Einstellung.
- Anwendungslatenz — Messen Sie die Latenz anhand einer Anwendungsmetrik, anhand strukturierter Protokolle oder anhand der Latenzmessungen Ihrer Haustür. Wenn die Latenz mit steigender Parallelität pro Umgebung zunimmt, reduzieren Sie `maxConcurrencyPerEnvironment` um Konflikte pro Umgebung zu reduzieren und die Last auf mehrere Umgebungen zu verteilen.

Skalierung für MultiRegion und PerRegion Endpunkte

Für Webfunktionen mit MultiRegion oder PerRegion Endpunkte, die Skalierungseinstellungen gelten jeweils unabhängig AWS-Region wo der Endpunkt arbeitet.

Jede Region erstellt und verwaltet ihren eigenen Pool von Ausführungsumgebungen. Der Verkehr in einer Region verbraucht nicht die Skalierungskapazität einer anderen Region.

Beispiele für Skalierung

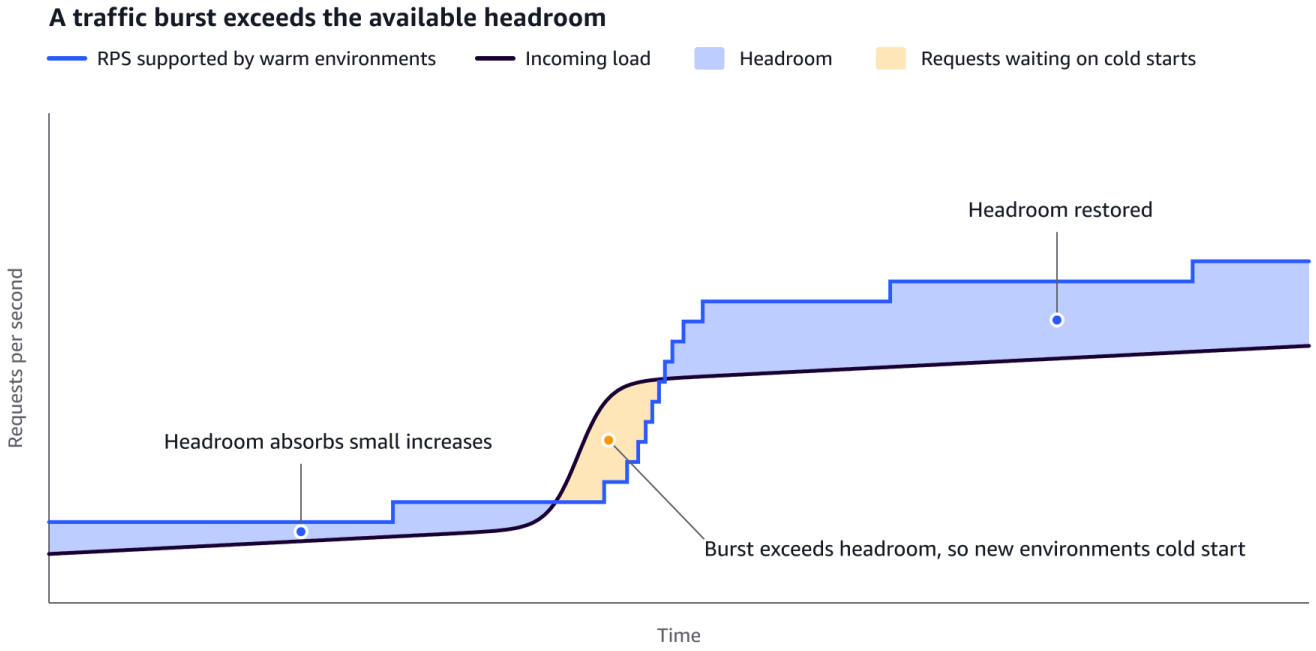
Lambda skaliert Ihren Endpunkt, um mehr Spielraum über der aktuellen Last zu haben. Headroom ist der Unterschied zwischen der Last eingehender Anfragen und der gesamten Anforderungslast, die warme Ausführungsumgebungen verarbeiten können. Es absorbiert kleine Datenverkehrszuwächse, ohne auf den Start neuer Umgebungen warten zu müssen. Die Kapazität wird schrittweise erhöht, da in jeder neuen Umgebung die Kapazität erst nach Abschluss der Initialisierung hinzugefügt wird. Wenn der Endpunkt ein Limit erreicht, hört die Kapazität auf zu wachsen, und Lambda drosselt Anfragen, die über das hinausgehen, was der Endpunkt akzeptieren kann.

Die folgenden Beispiele zeigen, wie der Endpunkt einen Datenverkehrsschub verarbeitet, wie sich das dynamische Parallelitätslimit pro Umgebung bei Auslastungstests für eine neue Webfunktion

ändert und wie sich der Endpunkt verhält, wenn die Last die einzelnen Grenzwerte erreicht. Die Diagramme veranschaulichen das allgemeine Verhalten und sind nicht maßstabsgetreu dargestellt.

Umgang mit einem Verkehrsschub

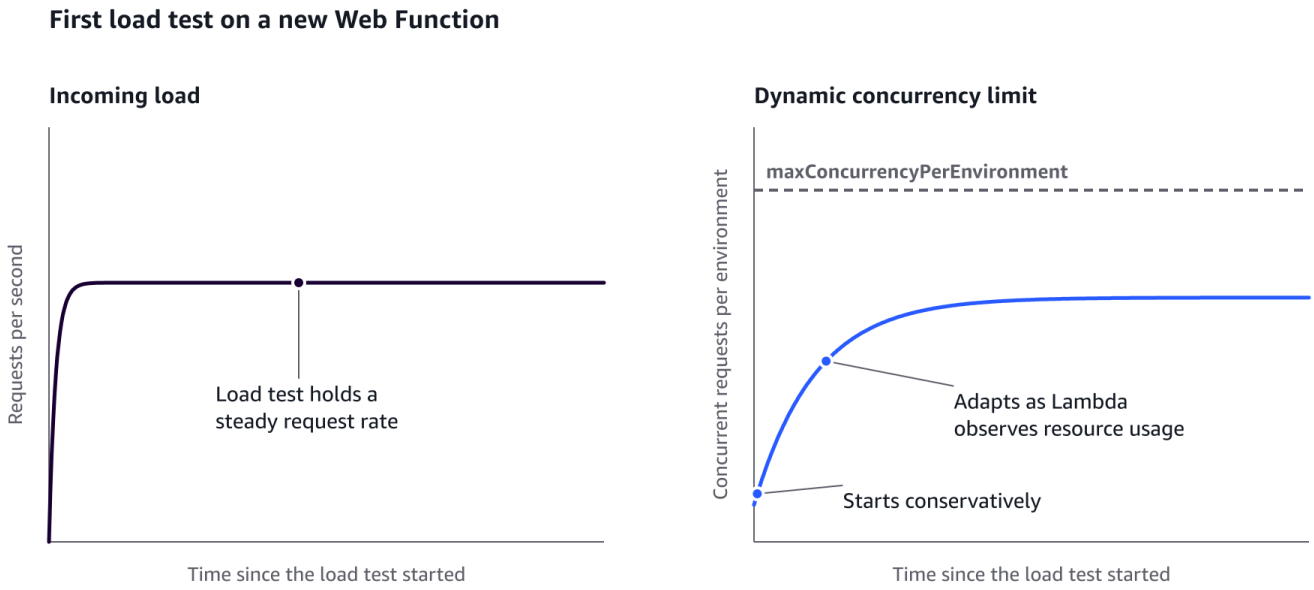
Wenn die Auslastung allmählich zunimmt, fängt der Headroom den Anstieg auf, während Lambda im Hintergrund neue Umgebungen skaliert. Wenn die Last schneller ansteigt, als der Headroom aufnehmen kann, können die warmen Umgebungen nicht alle eingehenden Anfragen verarbeiten. Lambda startet neue Umgebungen, um die Last zu bewältigen, die über dem Headroom liegt, und diese Anfragen warten, bis die neuen Umgebungen kalt gestartet werden. Sobald die neuen Umgebungen bereit sind, stellt Lambda den Headroom über der neuen Last wieder her.



Dynamisches Limit für die Parallelität einer neuen Webfunktion

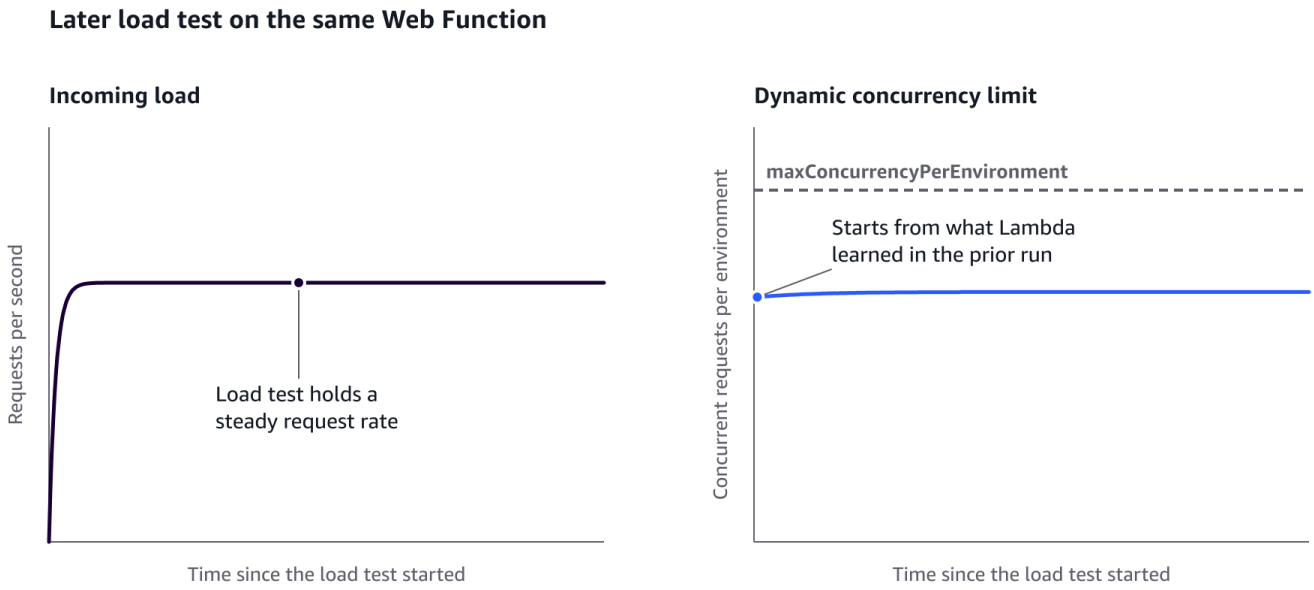
Das dynamische Parallelitätslimit pro Umgebung beginnt konservativ für eine neue Webfunktion. Während die Funktion den Datenverkehr verarbeitet, lernt Lambda aus ihrer Ressourcennutzung und passt das Limit an. Lambda merkt sich, was es gelernt hat, als die Funktion auf Null skaliert wurde, sodass Umgebungen, die später beginnen, vom früheren Datenverkehr profitieren.

Die folgenden Diagramme zeigen den ersten Belastungstest für eine neue Webfunktion. Die eingehende Last steigt schnell auf eine konstante Anforderungsrate an und hält diese. Das dynamische Limit für Parallelität beginnt niedrig, steigt, wenn Lambda es an die Ressourcenauslastung der Funktion anpasst, und pendelt sich wieder unter `maxConcurrencyPerEnvironment`.



Da Lambda sich daran erinnert, was es aus vorherigem Traffic gelernt hat, beginnen spätere Auslastungstests auf derselben Webfunktion nicht konservativ.

Die folgenden Diagramme zeigen einen späteren Belastungstest für dieselbe Webfunktion. Die eingehende Last steigt auf dieselbe konstante Anforderungsrate, und das dynamische Parallelitätslimit beginnt auf dem Niveau, das Lambda im vorherigen Lauf gelernt hat.

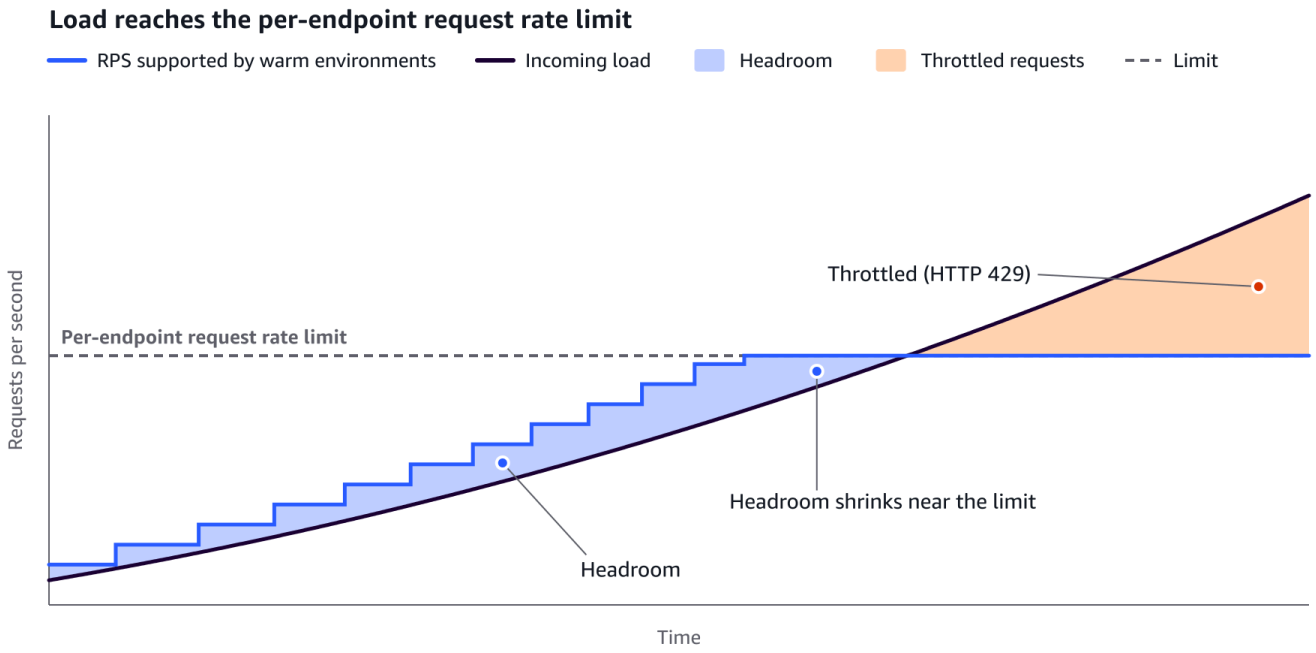


Da das Limit konservativ beginnt, verwendet der erste Auslastungstest für eine neue Webfunktion möglicherweise mehr Ausführungsumgebungen als bei späteren Ausführungen. Um die Leistung

zu messen, die Ihre Funktion bei gleichbleibendem Datenverkehr erwarten kann, führen Sie den Auslastungstest mehrmals aus und verwenden Sie die Ergebnisse späterer Läufe.

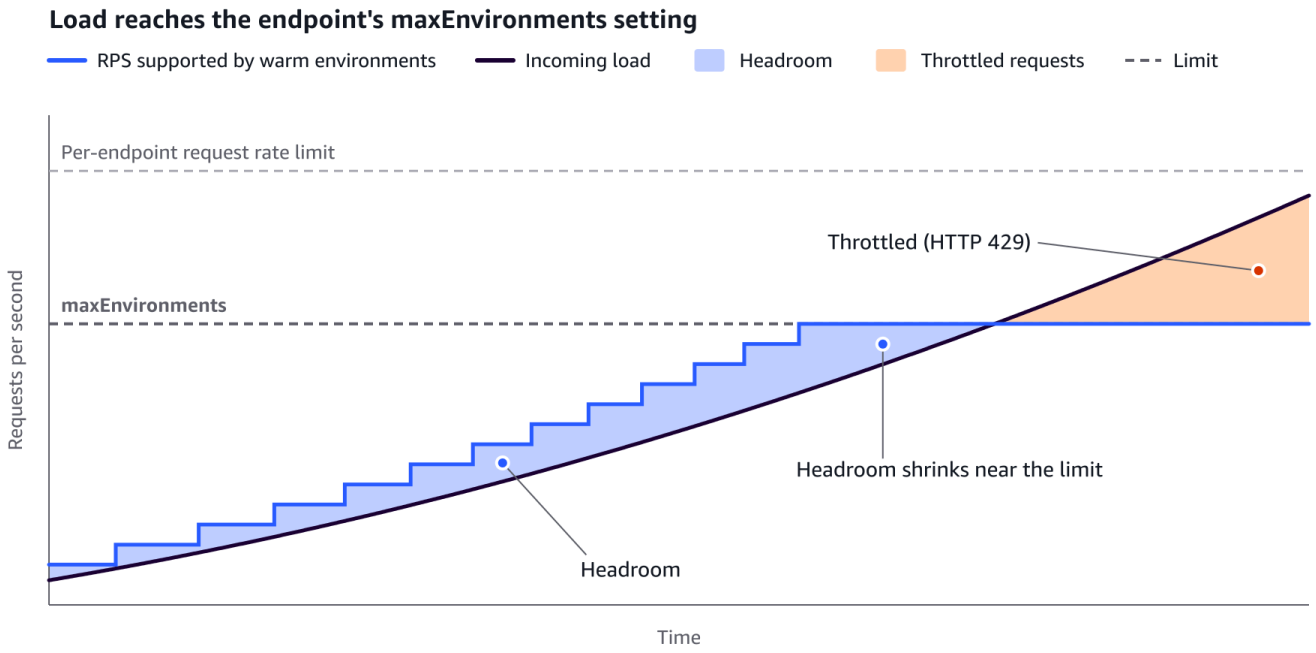
Das Limit der Anforderungsrate pro Endpunkt wird erreicht

Das Limit für die Anforderungsrate begrenzt, wie viel Last der Endpunkt aufnehmen kann, sodass die Kapazität nicht darüber hinauswächst. Wenn sich die Auslastung dem Preiskontingent pro Endpunkt nähert oder Ihr `rateLimit` Bei zunehmender Einstellung schrumpft die Kopffreiheit. Wenn die Last den Grenzwert überschreitet, gibt Lambda für Anfragen, die den Grenzwert überschreiten, HTTP 429 zurück. Um mehr Anfragen anzunehmen, erhöhen Sie `rateLimit`, bis zum Kontingent pro Endpunkt.



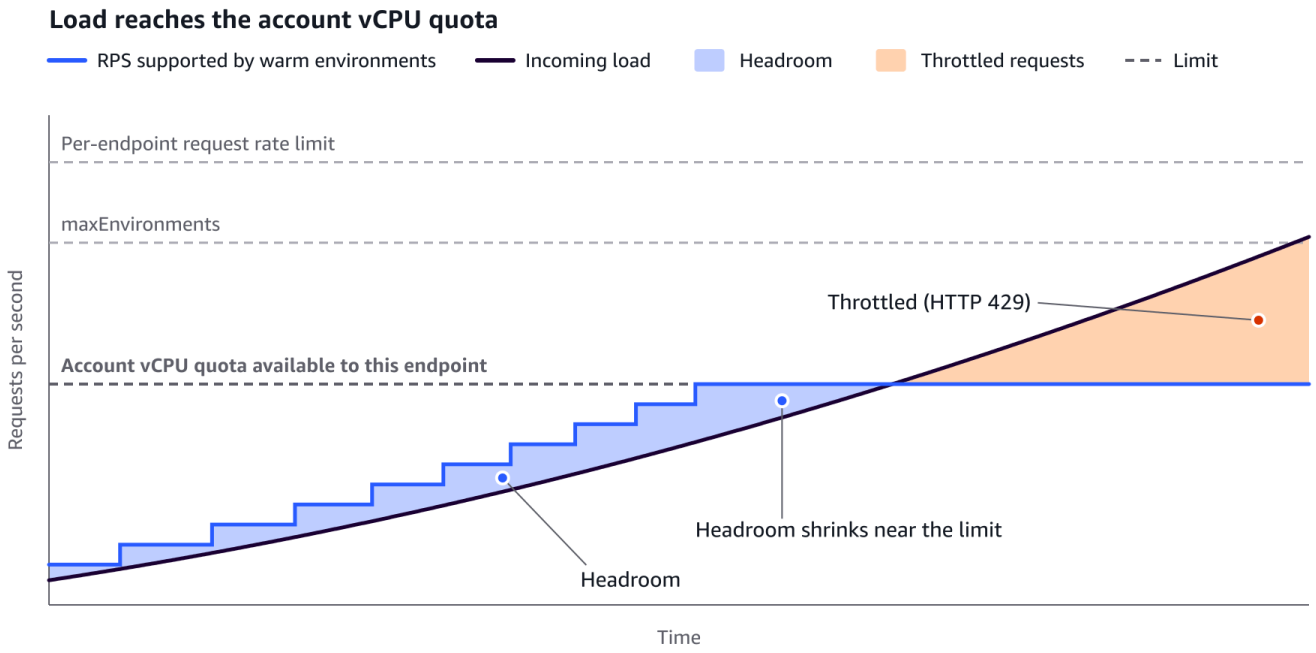
MaxEnvironments wird erreicht

In diesem Beispiel `maxEnvironments` ist niedriger als das Limit für die Anforderungsrate. Wenn der Endpunkt erreicht ist `maxEnvironments`, Lambda hört auf, Ausführungsumgebungen hinzuzufügen. Die Belastung kann weiter zunehmen, sodass die Kopffreiheit schrumpft. Wenn die Last das übersteigt, was die Umgebungen verarbeiten können, drosselt Lambda neue Anfragen mit HTTP 429, obwohl die Anforderungsrate immer noch unter dem Ratenlimit liegt. Um mehr Kapazität zu ermöglichen, erhöhen Sie `maxEnvironments`.



Das vCPU-Kontingent des Kontos wird erreicht

Alle Endpoints in einem Konto teilen sich jeweils das vCPU-Kontingent AWS-Region. Umgebungen, die von anderen Endpunkten verwendet werden, reduzieren die Skalierbarkeit dieses Endpunkts. In diesem Beispiel ist das für den Endpunkt verfügbare vCPU-Kontingent niedriger als das der beiden `maxEnvironments` Einstellung und ihr Limit für die Anforderungsrate. Wenn das Konto das Kontingent erreicht, stoppt Lambda das Hinzufügen von Umgebungen und drosselt Anfragen, die die verfügbare Kapazität überschreiten, obwohl der Endpunkt seine eigenen Grenzwerte nicht erreicht hat. Um mehr Kapazität zuzulassen, fordern Sie eine Erhöhung des vCPU-Kontingents an.



Lebenszyklus der Ausführungsumgebung

Eine Ausführungsumgebung ist eine isolierte Recheninstanz, auf der Ihr HTTP-Server ausgeführt wird. AWS Lambda erstellt Ausführungsumgebungen für Ihre Lambda-Webfunktion, wenn der Datenverkehr eingeht. Es leitet Anfragen an aktive Umgebungen weiter und entfernt Umgebungen, die nicht mehr benötigt werden. Wenn Sie diesen Lebenszyklus verstehen, können Sie die Startleistung optimieren und Code schreiben, der das Initialisieren und Herunterfahren korrekt handhabt.

Lebenszyklusphasen

Jede Ausführungsumgebung durchläuft die folgenden Phasen:



Phase	Was passiert	Bearbeitung von Anfragen
Initialisierung	Lambda lädt Ihren Code, bereitet die Laufzeit vor und startet Ihren HTTP-Serverprozess . Ihr Server muss `AWS_LAMBDA_HTTP_ENDPOINT` innerhalb von 10 Sekunden beginnen, die Adresse abzuhören.	Die Umgebung kann während der Initialisierung keine Anfragen bearbeiten.
Bereit	Ihr HTTP-Server hört zu und die Umgebung ist für den Empfang von Anfragen verfügbar. Lambda leitet eingehende Anfragen an diese Umgebung weiter.	Die Umgebung akzeptiert Anfragen bis zum `maxConcurrencyPerEnvironment` Limit.
Herunterfahren	Lambda beendet die Weiterleitung neuer Anfragen an die Umgebung, lässt laufende Anfragen nach bestem Wissen und Gewissen beenden`SIGTERM`, sendet und beendet dann die Umgebung.	Es werden keine neuen Anfragen an die Umgebung weitergeleitet. In-flight Anfragen können fortgesetzt werden, der Abschluss ist jedoch nicht garantiert. `SIGTERM`Danach hat der Vorgang bis zu 1 Sekunde Zeit, um die Bereinigung abzuschließen. Der Prozessstatus ist verloren gegangen.

Initialisierung (Kaltstart)

Wenn Lambda eine neue Ausführungsumgebung erstellt, umfasst die Initialisierungsphase die folgenden Schritte:

1. Lambda lädt den Code aus Ihrem ZIP-Bereitstellungspaket.
2. Lambda richtet die Laufzeitumgebung ein (einschließlich Node.js Ihrer Umgebungsvariablen).
3. Lambda startet Ihren Bewerbungsprozess.
4. Ihr HTTP-Server beginnt, auf dem Host und Port zu lauschen, die in der `AWS_LAMBDA_HTTP_ENDPOINT` Umgebungsvariablen angegeben sind. Er muss innerhalb von 10 Sekunden mit dem Abhören beginnen.

5. Lambda erkennt, dass Ihr Server zuhört, und markiert die Umgebung als bereit.

Ein Kaltstart tritt auf, wenn Lambda eine neue Umgebung erstellen muss, um eine Anfrage zu bearbeiten, da keine Umgebungen verfügbar sind. Bei der ersten Anfrage nach einem Kaltstart kommt es in der Initialisierungsphase zu zusätzlicher Latenz. Bei nachfolgenden Anfragen an dieselbe Umgebung fallen diese Kosten nicht an.

Um die Auswirkungen des Kaltstarts zu reduzieren:

- Minimieren Sie die Größe Ihres Bereitstellungspakets. Kleinere Pakete werden schneller geladen.
- Reduzieren Sie die Serverstartzeit. Verschieben Sie umfangreiche Initialisierungen (z. B. das Laden großer Datendateien) auf die Zeit, nachdem der Server mit der Überwachung begonnen hat. Lambda leitet Anfragen an die Umgebung weiter, sobald der Server zuhört. Lassen Sie also Anforderungshandler, die von der verzögerten Arbeit abhängen, warten, bis sie abgeschlossen ist, indem sie beispielsweise auf ein gemeinsames Versprechen warten.
- Verwenden Sie die verzögerte Initialisierung für Datenbankverbindungen und externe Service-Clients. Stellen Sie Verbindungen bei der ersten Anfrage und nicht beim Start her.

Scale-down und Herunterfahren

Lambda überwacht die Umgebungsauslastung Ihres Endgeräts und entfernt nicht benötigte Umgebungen, z. B. wenn der Traffic abnimmt. Vor dem Entfernen einer Umgebung verwendet Lambda die folgende Shutdown-Sequenz:

1. Lambda beendet die Weiterleitung neuer Anfragen an die Umgebung.
2. Lambda ermöglicht es, Anfragen während des Fluges nach bestem Wissen und Gewissen zu bearbeiten.
3. Lambda sendet`SIGTERM`. Der Vorgang hat bis zu 1 Sekunde Zeit, um die Bereinigung abzuschließen, bevor Lambda ihn beendet und die Umgebung entfernt.

Wenn kein Verkehr vorhanden ist, skaliert Lambda Ihre Webfunktion auf Umgebungen ohne. Die nächste Anfrage nach einer Phase der Skalierung auf Null löst einen neuen Kaltstart aus.

Wenn Ihre Anwendung während des Herunterfahrens eine Bereinigung durchführen muss, tun Sie das. `SIGTERM` Die Bereinigung muss innerhalb von 1 Sekunde abgeschlossen sein. Verlassen Sie sich nicht darauf, dass Shutdown-Callbacks ausgeführt werden, da Lambda nicht in allen Fehlermodi,

z. B. bei einem zugrunde liegenden Hardwarefehler, ein ordnungsgemäßes Herunterfahren garantieren kann.

Das folgende Beispiel behandelt: `SIGTERM`

```
process.on('SIGTERM', () => {
  // Finish short cleanup, such as flushing buffered logs, within 1 second.
  process.exit(0);
});
```

Wiederverwendung der Umgebung

Lambda verwendet Ausführungsumgebungen für mehrere Anfragen wieder, um wiederholte Kaltstarts zu vermeiden. Eine einzelne Umgebung kann während ihrer Lebensdauer Tausende von Anfragen bearbeiten. Das bedeutet Folgendes:

- Globaler Status bleibt bestehen — Variablen, Caches und Verbindungen, die Ihr Server beim Start initialisiert, bleiben auch bei allen Anfragen bestehen. Sie können dies für In-Memory-Caching oder Verbindungspooling verwenden.
- Dateisystemstatus bleibt bestehen — Dateien, die Ihr Code in das lokale Dateisystem schreibt (z. B.`/tmp`), bleiben bestehen, bis Lambda die Umgebung entfernt.
- Keine garantierte Lebensdauer — Lambda kann eine Umgebung jederzeit entfernen. Verlassen Sie sich nicht auf eine Umgebung, die über einen bestimmten Zeitraum andauert. Speichern Sie wichtige Daten in einem externen Speicher wie DynamoDB oder Amazon S3.

Gleichzeitige Bearbeitung von Anfragen

Jede Ausführungsumgebung für Webfunktionen verarbeitet mehrere gleichzeitige HTTP-Anfragen. Die `maxConcurrencyPerEnvironment` Einstellung in der Revision steuert die maximale Anzahl gleichzeitiger Anfragen pro Umgebung.

Da Ihr Server mehrere Anfragen gleichzeitig verarbeitet, muss Ihr Code für die gleichzeitige Verwendung sicher sein:

- Speichern Sie den Status pro Anfrage nicht in globalen Variablen. Verwenden Sie stattdessen Variablen mit Anforderungsbereich oder einen Middleware-Kontext.
- Schützt gemeinsam genutzte veränderbare Zustände wie Caches oder Zähler, die Sie grenzübergreifend ändern. `await` Anfragen verschachteln sich in der Ereignisschleife. Wenn sich

eine Sequenz mit Lesen, Ändern und Schreiben über einen asynchronen Vorgang erstreckt, kann eine weitere aktive Anforderung den Status lesen oder ändern, bevor die Sequenz abgeschlossen ist.

- Beachten Sie, dass dabei eine Ereignisschleife mit einem einzigen Thread verwendet wird. Node.js CPU-intensive synchrone Operationen blockieren alle gleichzeitigen Anfragen in dieser Umgebung. Verwenden Sie asynchron I/O und vermeiden Sie es, die Ereignisschleife zu blockieren.

Weitere Informationen zu Parallelitätseinstellungen und Optimierung finden Sie unter. the section called “Skalierung”

Laufzeiten der Lambda-Webfunktion

Lambda Web Functions führen Ihren HTTP-Servercode mithilfe verwalteter Laufzeiten aus. Ihre Anwendung muss einen HTTP-Server starten, der den in der Umgebungsvariablen angegebenen Host und Port überwacht. `AWS_LAMBDA_HTTP_ENDPOINT` Lambda-Webfunktionen werden derzeit unterstütztNode.js. Standardmäßig wird die Runtime `index.js` im Stammverzeichnis des Bereitstellungspakets geladen. Um eine andere Eintragsdatei zu verwenden, legen Sie die `AWS_LAMBDA_NODEJS_ENTRYPOINT` Umgebungsvariable fest. Details hierzu finden Sie unter the section called “Umgebungsvariablen”.

Unterstützt Node.js Versionen

Laufzeit-ID	Node.js-Version	Status
`nodejs24.x`	Node.js24	Unterstützt

Geben Sie die Laufzeit-ID in dem `runtimeConfig` Abschnitt an, wenn Sie eine Revision erstellen. Beispiel:

```
"runtimeConfig": { "runtime": "nodejs24.x" }
```

Die `nodejs24.x` Web Function-Laufzeit folgt demselben Verfallszyklus wie die entsprechende Lambda Event Functions-Laufzeit. Informationen zu den Verfallsdaten und zum Ablauf der Verwarnung finden Sie unter. the section called “Richtlinie für den Laufzeitablauf”

Automatisches Runtime-Patching

Lambda patcht die Node.js Laufzeit automatisch mit Sicherheitsupdates und Bugfixes. Das Patchen bezieht sich auf die zugrunde liegende Laufzeitebene und ändert Ihren Anwendungscode nicht. Die `runtime` ID Ihrer Revision (z. B.`nodejs24.x`) entspricht immer der neuesten gepatchten Version dieser Haupt-Runtime.

> **Note**
>
> Steuerelemente zur Laufzeitverwaltung werden für Webfunktionen nicht unterstützt. Sie können sich nicht an eine bestimmte Patch-Version binden oder das automatische Patchen deaktivieren. Lambda verwendet immer die neueste gepatchte Version der Laufzeit, die Sie angegeben haben.

Weitere Informationen darüber, wie Lambda Runtime-Updates verwaltet, finden Sie unterthe section called “Aktualisierungen der Laufzeitversion”.

Unterstützte Web-Frameworks

Sie können jedes Node.js Web-Framework verwenden, das einen HTTP-Server startet. Zu den beliebten Frameworks gehören:

- Express
- Next.js
- Fastify
- Koa

Sie benötigen den AWS Lambda Web Adapter oder einen anderen Event-zu-HTTP-Adapter nicht. Ihre Anwendung verarbeitet HTTP-Anfragen direkt mithilfe des Standard-Routing und der Middleware des Frameworks. Einige Frameworks, z. B. Next.js im Standalone-Modus, lesen ihre eigenen Umgebungsvariablen für die Listen-Adresse. Verwenden Sie für diese Frameworks eine `index.js` Eintragsdatei, die diese Variablen festlegt, `AWS_LAMBDA_HTTP_ENDPOINT` bevor der Server gestartet wird. Details hierzu finden Sie unter the section called “Umgebungsvariablen”. Eine vollständige Anwendung finden Sie unterthe section called “Minimal Express Anwendung”.

Node.js Ausführungsmodell

Node.jsverwendet ein ereignisgesteuertes Ausführungsmodell. Node.jsFührt Ihren JavaScript Code standardmäßig in einem einzigen Haupt-Thread aus, und I/O Operationen (Netzwerkaufrufe, Dateisystemzugriff) sind nicht blockierend und ereignisgesteuert. Ihre Anwendung kann bei Bedarf zusätzliche Threads mit dem `worker_threads` Modul erstellen. Dieses Modell hat wichtige Auswirkungen auf Webfunktionen:

Parallelität durch Async I/O

Ihr Server verarbeitet mehrere gleichzeitige Anfragen mithilfe asynchroner Operationen (`async`/`await`, Promises, Callbacks). Während eine Anfrage auf eine Datenbankantwort wartet, bedient die Event-Schleife andere Anfragen. Aus diesem Grund können Web Functions viele gleichzeitige Anfragen in einer einzigen Umgebung ohne mehrere Threads verarbeiten.

Vermeiden Sie es, die Ereignisschleife zu blockieren

CPU-intensive synchrone Operationen (umfangreiche JSON-Analyse, komplexe Berechnungen, synchrone Dateilesevorgänge) blockieren die Ereignisschleife und verhindern, dass alle anderen Anfragen verarbeitet werden. Wenn Ihre Anwendung CPU-heavy Arbeit verrichtet, verlagern Sie sie auf Worker-Threads oder reduzieren `maxConcurrencyPerEnvironment` Sie sie, um die Auswirkungen zu begrenzen.

Isolierung beantragen

Da alle Anfragen denselben Prozess und dieselbe Ereignisschleife verwenden, sollten Sie vermeiden, den Status pro Anfrage in globalen Variablen zu speichern. Verwenden Sie einen Middleware-Kontext, der sich auf Anfragen bezieht (z. B. Express `req` `res` /objects), um Anforderungsdaten zu isolieren.

Hinweise zur Konfiguration auf der `maxConcurrencyPerEnvironment` Grundlage Ihrer Anwendungs- und CPU-Eigenschaften finden Sie unter. I/O the section called “Skalierung”

Speicher und Node.js Haufen

Jede Ausführungsumgebung hat 2 GB Arbeitsspeicher. Für den Node.js V8-Heap gilt Folgendes:

- Der Standard-Heap ist 1.024 MB groß.

- Um einen größeren Heap zu verwenden, geben Sie `NODE_OPTIONS` beispielsweise die Umgebungsvariablen der Revision ein `NODE_OPTIONS=--max-old-space-size=1536` (der Wert ist in MB angegeben).
- Ein Wert, den Sie eingegeben haben, `NODE_OPTIONS` ersetzt den Standardwert. Wenn Sie nur festlegen`--max-old-space-size`, legt Lambda nichts fest `--max-semi-space-size` und Node.js verwendet dafür seine eigene Standardeinstellung.
- Der Heap, die Puffer, die systemeigenen Zuweisungen, die Laufzeit und die gleichzeitigen Anfragen müssen alle in den 2 GB Umgebungsspeicher passen.

So leitet Lambda Anfragen an Ihre Anwendung weiter

Webfunktions-Endpunkte akzeptieren nur HTTPS-Verkehr auf Port 443. Klares HTTP wird nicht bereitgestellt, sodass Sie HTTP nicht zu HTTPS umleiten oder TLS in Ihrer Anwendung konfigurieren müssen.

Lambda beendet TLS am Endpunkt und leitet jede Anfrage an den HTTP-Server Ihrer Anwendung innerhalb der Ausführungsumgebung weiter. Ihre Anwendung stellt reines HTTP an der Adresse bereit, die Lambda bereitstellt. Sie verarbeiten in Ihrem Code niemals Zertifikate oder TLS-Konfigurationen.

Lambda wählt den Host und den Port aus, auf die Ihr Server warten muss. Lambda übermittelt Ihrer Anwendung die Adresse über die `AWS_LAMBDA_HTTP_ENDPOINT` Umgebungsvariable im `host:port` Format. Verwenden Sie diese Variable, die Lambda festlegt, um zu bestimmen, wo Ihr Server lauscht. Ihre Anwendung muss die Variable beim Start lesen:

```
const [host, port] = (process.env.AWS_LAMBDA_HTTP_ENDPOINT ||
 'localhost:3000').split(':');
app.listen(port, host);
```

Lambda legt die Variable nicht auf Ihrem lokalen Computer fest. Die Fallback-Adresse gilt nur, wenn die Variable nicht gesetzt ist, sodass sie nur in lokalen Läufen verwendet wird. Wenn Sie eine andere lokale Adresse verwenden möchten, legen Sie diese selbst fest`AWS_LAMBDA_HTTP_ENDPOINT`. Ein Beispiel finden Sie unter the section called “Führen Sie den HTTP-Server lokal aus”.

Lambda fügt jeder Anfrage, die es an Ihre Anwendung weiterleitet, einen `lambda-web-request-id` Header hinzu und gibt denselben Wert wie ein Antwortheader zurück, sodass Sie eine Client-Anfrage mit Ihren Anwendungsprotokollen korrelieren können.

Lambda leitet Webanfragen an die von `AWS_LAMBDA_HTTP_ENDPOINT` angegebene Adresse weiter. Wenn Ihr Server auf einem anderen Host oder Port lauscht, erreichen Anfragen Ihre Anwendung nicht.

Ihr Server muss innerhalb von 10 Sekunden nach dem Start mit der Überwachung beginnen, und jede Anfrage, die er bearbeitet, ist an die `timeoutSeconds` Einstellung der Revision gebunden. Informationen zur Reihenfolge beim Starten und Herunterfahren finden Sie unterthe section called “Lebenszyklus der Umgebung”. Informationen zur Konfiguration finden `timeoutSeconds` Sie unterthe section called “Überarbeitungen”.

Konfiguration von Lambda-Webfunktionen

Sie konfigurieren die meisten Web-Funktionseinstellungen in einer Revision, bei der es sich um eine unveränderliche Momentaufnahme Ihres Anwendungscodes und Ihrer Konfiguration handelt. Da eine Revision unveränderlich ist, bedeutet das Ändern einer Revisionseinstellung, dass eine neue Revision mit den aktualisierten Werten erstellt wird. Sie legen Tags und die ressourcenbasierte Richtlinie in der Webfunktion fest. Ihre Revisionen und Endpunkte erben ihre Tags, und die ressourcenbasierte Richtlinie legt fest, wer auf die Webfunktion, ihre Revisionen und ihre Endpunkte einwirken kann. Die Endpunkteinstellungen gelten für jeden Endpunkt, einschließlich Authentifizierungstyp, automatischer Bereitstellungsmodus, Revisionsgewichtung, Skalierung und Drosselung. Sie können diese ändern, ohne eine neue Revision zu erstellen. In den folgenden Themen werden die Einstellungen beschrieben, die Sie konfigurieren können.

Themen

- Bereitstellungspaket für Webfunktionen
- Befehlssatz-Architektur
- Anforderungstimeout
- Umgebungsvariablen
- Webfunktionen taggen
- Verwenden des Metadaten-Endpunkts

Bereitstellungspaket für Webfunktionen

Sie verpacken Ihren Web Function Code als ZIP-Datei und speichern ihn in einem versionierten Amazon S3 S3-Bucket. Die Revision `codeConfig` verweist auf den Bucket, den Schlüssel und optional auf eine Objektversion. Wenn Sie die Version weglassen, verwendet Lambda die neueste

Version zum Zeitpunkt der Erstellung, sodass jede Revision auf eine bestimmte Version Ihres Pakets verweist. Durch das Hochladen eines neuen Pakets auf denselben Schlüssel werden bestehende Revisionen nicht geändert, und Sie können zu einer früheren Revision zurückkehren, solange die Objektversion im Bucket verbleibt.

Das Paket muss die Eingabedatei Ihrer Anwendung und all ihre Abhängigkeiten enthalten (z. B.`node_modules`). Standardmäßig wird Lambda als Eintragsdatei `index.js` im Paketstamm geladen und ausgeführt, um Ihren HTTP-Server zu starten. Um einen anderen Namen oder Pfad zu verwenden (z. B.`dist/index.js`), legen Sie die `AWS_LAMBDA_NODEJS_ENTRYPOINT` Umgebungsvariable fest; siehethe section called “Umgebungsvariablen”.

Informationen zum Erstellen und Hochladen des Pakets auf Amazon S3 sowie zum Erstellen einer Revision, die darauf verweist, finden Sie unter the section called “Kodieren Sie in Amazon S3” undthe section called “Erstellen Sie Ihre erste Webfunktion”.

Befehlssatz-Architektur

Webfunktionen werden ausschließlich auf der Arm64-Architektur ausgeführt, die von Graviton-Prozessoren angetrieben AWS wird. Im Gegensatz zu Lambda Event Functions können Sie den Befehlssatz nicht auswählen: Es muss keine Architektureinstellung konfiguriert werden.

Jede Ausführungsumgebung für Webfunktionen verwendet 2 Arm64-vCPUs. Ein regionsspezifisches Kontingent begrenzt die Arm64-vCPU-Kapazität Ihres Kontos. Informationen zu dem Kontingent finden Sie unter the section called “Lambda-Web-Funktionen”. Die Nutzungsmetriken, die Ihren Verbrauch im Vergleich dazu zeigen, finden Sie unter. the section called “Regionale Nutzungs- und Drosselungsmetriken”

Erstellen Sie native Abhängigkeiten für Linux auf arm64 (aarch64). Reiner Node.js JavaScript Code muss nicht geändert werden. Jede Abhängigkeit mit nativen Binärdateien muss einen Linux-Arm64-Build bereitstellen. Erstellen Sie die Abhängigkeit auf einer Linux-Arm64-Maschine oder in einem Linux-Arm64-Container oder kompilieren Sie sie für Linux-Arm64 kreuzkompilieren. Ein Beispiel, das Abhängigkeiten in einem Arm64-Container installiert, finden Sie unter. the section called “Testen Sie das Bereitstellungspaket”

Anforderungstimeout

Die `timeoutSeconds` Einstellung steuert die maximale Zeit, die eine einzelne HTTP-Anfrage ausgeführt werden kann, bevor Lambda sie beendet. Gültige Werte sind 3—900 Sekunden (15 Minuten). Standardmäßig ist ein Zeitraum von 30 Sekunden festgelegt.

Wählen Sie einen Timeout-Wert, der auf der am längsten erwarteten Anfrage Ihrer Anwendung basiert. Wenn Sie das Timeout zu niedrig einstellen, schlagen Anfragen mit langer Laufzeit fehl. Wenn Sie es zu hoch einstellen, werden festgefahrene Anfragen länger ausgeführt, bevor Lambda sie beendet.

Wenn eine Nicht-Streaming-Anfrage das Timeout überschreitet, gibt Lambda HTTP 504 mit dem Header von zurück. `lambda-web-error-type` `Runtime.RequestTimeout` Bei einer Streaming-Antwort begrenzt das Timeout den gesamten Stream. Wenn das Timeout abgelaufen ist, beendet Lambda den Stream und der Client erhält den bereits gesendeten Statuscode, normalerweise 200, mit einem abgeschnittenen Hauptteil und ohne Header. `lambda-web-error-type`

Umgebungsvariablen

Umgebungsvariablen sind Schlüssel-Wert-Paare, die zur Laufzeit für Ihren Anwendungscode verfügbar sind. Verwenden Sie Umgebungsvariablen für nicht sensible Konfigurationen, die Sie ändern möchten, ohne den Code zu ändern, z. B. Feature-Flags, Protokollebenen und Ressourcenbezeichner wie einen Datenbank-Hostnamen. Speichern Sie sensible Werte wie API-Schlüssel und Datenbankanmeldedaten nicht in Umgebungsvariablen. Verwenden Sie stattdessen Secrets Manager oder AWS Systems Manager Parameter Store und rufen Sie die Werte zur Laufzeit ab.

Umgebungsvariablen werden bei der Erstellung der Revision festgelegt und sind nach der Erstellung unveränderlich. Um eine Umgebungsvariable zu ändern, erstellen Sie eine neue Revision. Geben Sie in der neuen Version jede Umgebungsvariable und jede `serviceConfig` Einstellung an, die Sie beibehalten möchten, nicht nur die Werte, die sich ändern. Einstellungen, die Sie auslassen, erhalten ihre Standardwerte, und Umgebungsvariablen, die Sie auslassen, sind nicht festgelegt.

Lambda legt die folgende Umgebungsvariable speziell für Webfunktionen fest:

- **`AWS_LAMBDA_HTTP_ENDPOINT`**— Der Host und der Port, auf die Ihr HTTP-Server warten muss, im `host:port` Format. Lambda setzt diese Variable und leitet Anfragen an diese Adresse weiter. Lesen Sie es in Ihrem Anwendungscode und binden Sie genau diesen Host und Port. Sie können diese Variable nicht in den Umgebungsvariablen einer Revision setzen. Binden Sie immer an die Adresse in dieser Variablen, nicht `localhost` an einen fest codierten Port. Die Runtime wartet darauf, dass Ihr Server eine Verbindung an dieser Adresse akzeptiert, und die Initialisierung schlägt fehl, falls dies nie der Fall ist. Verwenden Sie eine Fallback-Adresse nur für lokale Läufe. In einer Webfunktion legt Lambda immer diese Variable fest.

Webfunktionen empfangen auch andere Lambda-Laufzeitumgebungsvariablen wie `AWS_LAMBDA_FUNCTION_MEMORY_SIZEAWS_LAMBDA_METADATA_API`, und`AWS_LAMBDA_METADATA_TOKEN`. Eine allgemeine Liste finden Sie unterDefinierte Laufzeitumgebungsvariablen. Wenn Ihr Code oder eine Bibliothek eine Variable aus dieser Liste liest, stellen Sie sicher, dass Lambda sie in einer Webfunktion festlegt, bevor Sie sich darauf verlassen. Webfunktionen empfangen nicht jede Variable in dieser Liste.

Die folgenden Namen von Umgebungsvariablen sind für Lambda-Webfunktionen reserviert und können nicht festgelegt werden. Wenn Sie eine angeben, lehnt Lambda die Anforderung ab, eine Webfunktion oder Revision zu erstellen:

- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SECRET_KEY`, `AWS_SECURITY_TOKEN`, und `AWS_SESSION_TOKEN`
- `AWS_DEFAULT_REGION` und `AWS_REGION`
- `AWS_LAMBDA_HTTP_ENDPOINT`, `AWS_LAMBDA_RUNTIME_API`, `AWS_LAMBDA_INITIALIZATION_TYPE`, und `AWS_LAMBDA_FUNCTION_MEMORY_SIZE`
- `AWS_LAMBDA_METADATA_API` und `AWS_LAMBDA_METADATA_TOKEN`

Einige Frameworks lesen stattdessen ihre eigenen Umgebungsvariablen für die Listen-Adresse. Zum Beispiel liest der Next.js Standalone-Modus `HOSTNAME` und`PORT`. Verwenden Sie für solche Frameworks eine `index.js` Eintragsdatei, die diese Variablen festlegt, `AWS_LAMBDA_HTTP_ENDPOINT` bevor Sie den Server starten.

Standardmäßig wird die Runtime `index.js` im Stammverzeichnis Ihres Bereitstellungspakets als Eingabedatei Ihrer Anwendung geladen. Die Runtime liest den Einstiegspunkt nicht von`package.json`. Wenn Ihre Eintragsdatei einen anderen Namen oder Pfad hat (z. B.`dist/index.js`), setzen Sie die `AWS_LAMBDA_NODEJS_ENTRYPOINT` Umgebungsvariable auf ihren Pfad relativ zum Stammverzeichnis Ihres Pakets.

Das folgende Beispiel zeigt, wie die `AWS_LAMBDA_HTTP_ENDPOINT` Umgebungsvariable in einer Node.js Anwendung gelesen wird:

```
// Lambda provides the address to bind in AWS_LAMBDA_HTTP_ENDPOINT ("host:port").
// The fallback applies only when running locally.
const [host, port] = (process.env.AWS_LAMBDA_HTTP_ENDPOINT ||
 'localhost:3000').split(':');
app.listen(port, host, () => {
  console.log(`Server listening on ${host}:${port}`);
```

```
});
```

Um Ihre eigenen Umgebungsvariablen festzulegen, fügen Sie dem `serviceConfig` Objekt eine `environmentVariables` Map hinzu, wenn Sie eine Revision erstellen, wie im folgenden Beispiel. Den vollständigen Befehl finden Sie unterthe section called “Eine Revision erstellen”.

```
{
  "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role",
  "environmentVariables": {
    "DATABASE_URL": "my-database-endpoint",
    "LOG_LEVEL": "info",
    "FEATURE_FLAG_NEW_UI": "true"
  }
}
```

Wenn Sie einen vom Kunden verwalteten Schlüssel angeben, verschlüsselt Lambda die ruhenden Umgebungsvariablen mit diesem Schlüssel. Weitere Informationen finden Sie unter the section called “Verschlüsselung im Ruhezustand”.

Webfunktionen taggen

Tags sind Schlüsselwertbezeichnungen, die Sie einer Webfunktion zuordnen, um Ihre Ressourcen zu organisieren und den Zugriff zu kontrollieren. Tags gelten für die Webfunktions-Ressource.

Um beim Erstellen einer Webfunktion Tags hinzuzufügen, verwenden Sie den `tags` Parameter von`CreateWebFunction`. Um Tags später hinzuzufügen, zu entfernen oder aufzulisten, verwenden Sie die `ListTags` Operationen `TagResource``UntagResource`, und.

Sie können Tags auf folgende Weise verwenden:

- Organisation und Suche — Gruppieren und suchen Sie Webfunktionen nach Attributen wie Projekt, Umgebung oder Team. Sie können eine Webfunktion anhand ihrer Tags nur in ihrer Heimatregion finden. Weitere Informationen finden Sie unter the section called “Sichtbarkeit von Replikaten”.
- Attribute-based Zugriffskontrolle (ABAC) — Schreiben Sie Richtlinien AWS Identity and Access Management (IAM), die Aktionen auf der Grundlage von Tags zulassen oder verweigern. Wird `aws:ResourceTag/${TagKey}` für Aktionen in einer Webfunktion und ihren Revisionen und Endpunkten verwendet, die die Tags der Webfunktion erben. Bei aktivierter `CreateWebFunction` Option können Sie auch und verwenden. `aws:RequestTag/${TagKey}` `aws:TagKeys` Sie können Revisionen oder Endpunkte nicht direkt taggen. Weitere Informationen erhalten Sie unter the section called “Berechtigungen” und the section called “Zugriffskontrolle”.

Für Tags gelten zwei Anforderungen:

- Tag-Schlüssel bestehen aus 1—128 Zeichen.
- Tag-Werte sind 0—256 Zeichen lang.
- Schlüssel und Werte können Buchstaben, Zahlen, Leerzeichen und die folgenden Zeichen enthalten. `_ . : / = + - @`

Verwenden des Metadaten-Endpunkts

Der Lambda-Metadaten-Endpunkt ist eine lokale HTTP-API innerhalb der Ausführungsumgebung. Sie gibt Metadaten in einem einfachen JSON-Format zurück, einschließlich der Availability Zone (AZ) -ID, in der die Umgebung ausgeführt wird. Verwenden Sie es, um zu Ressourcen mit derselben AZ wie Amazon ElastiCache und Amazon RDS weiterzuleiten, um die Latenz zu verringern. Sie können es auch verwenden, um AZ-aware Resilienzmuster zu implementieren.

> **Note**
>
> Eine Webfunktion führt einen persistenten Webserver und nicht einen Handler pro Aufruf aus. Lesen Sie den Metadaten-Endpunkt einmal beim Start der Anwendung und speichern Sie den Wert im Cache oder lesen Sie ihn pro Anfrage, wenn Sie ihn im Anforderungskontext benötigen. Gehen Sie nicht von einem neuen Handler pro Aufruf aus. Bei einer Webfunktion mit mehreren Regionen meldet der Endpunkt die AZ innerhalb der Region AWS-Region , die die Anfrage bearbeitet.

Sections

- Erste Schritte

Erste Schritte

Powertools for AWS Lambda bietet ein Hilfsprogramm für den Zugriff auf den Lambda-Metadaten-Endpunkt. Das Hilfsprogramm speichert die Antwort nach dem ersten Anruf im Cache. Weitere Informationen finden Sie unter Metadaten in der Dokumentation zu Powertools for AWS Lambda (TypeScript).

Da Web Functions auf läuftNode.js, verwendet das folgende Beispiel das Metadaten-Hilfsprogramm Powertools for AWS Lambda, um die AZ-ID einmal beim Serverstart zu lesen und sie für die gesamte

Lebensdauer des Prozesses zwischenzuspeichern. Führen Sie den folgenden npm-Befehl aus, um das Powertools-Paket und Express zu installieren:

```
npm install @aws-lambda-powertools/commons express
```

Das folgende Beispiel dient der Express Veranschaulichung. Derselbe Ansatz funktioniert mit jedem Node.js Web-Framework; siehethe section called “Unterstützte Web-Frameworks”. Lesen Sie die AZ-ID beim Start und verwenden Sie den zwischengespeicherten Wert bei der Bearbeitung von Anfragen erneut:

Example Abrufen der AZ-ID mit Powertools beim Start () TypeScript

```
import express from 'express';
import { getMetadata } from '@aws-lambda-powertools/commons/utils/metadata';

const app = express();

// Read the AZ ID once at startup and cache it for the life of the server.
const { AvailabilityZoneID: azId } = await getMetadata();

// Lambda provides the address to bind in AWS_LAMBDA_HTTP_ENDPOINT ("host:port").
// The fallback applies only when running locally.
const [host, port] = (process.env.AWS_LAMBDA_HTTP_ENDPOINT ||
 'localhost:3000').split(':');

app.get('/', (req, res) => {
  res.json({ azId });
});

app.listen(port, host, () => {
  console.log(`Server listening on ${host}:${port}, AZ: ${azId}`);
});
```

Sie können den Metadaten-Endpunkt auch direkt aufrufen. Lambda legt automatisch die Umgebungsvariablen `AWS_LAMBDA_METADATA_API` und die `AWS_LAMBDA_METADATA_TOKEN` Umgebungsvariablen in jeder Ausführungsumgebung fest, und Sie fügen das Token mit dem Bearer-Schema in den `Authorization` Header ein. Informationen zur Endpunkt-URL, zu den Anfrage- und Antwortformaten sowie zu Fehlerantworten finden Sie unter. the section called “Metadaten-Endpunkt”

Lesen Sie beispielsweise die Metadaten-Serveradresse von `AWS_LAMBDA_METADATA_API` und senden Sie eine authentifizierte GET-Anfrage:

```
METADATA_ENDPOINT="http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-
environment"
RESPONSE=$(curl -s -H "Authorization: Bearer ${AWS_LAMBDA_METADATA_TOKEN}"
 "$METADATA_ENDPOINT")
AZ_ID=$(echo "$RESPONSE" | jq -r '.AvailabilityZoneID')
```

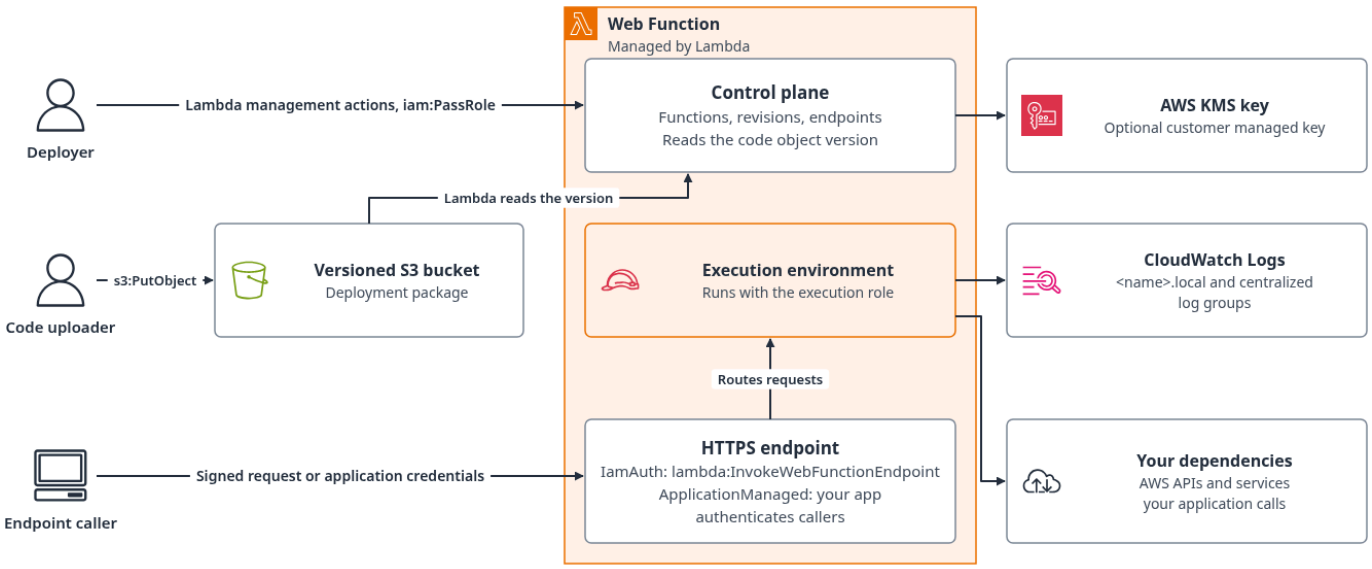
Der Endpunkt kehrt zurück`application/json`. Beispiel: `{"AvailabilityZoneID": "use1-az1"}`.

Berechtigungen für Lambda-Webfunktionen

Lambda Web Functions verwenden AWS Identity and Access Management (IAM), um zu steuern, was Ihr Anwendungscode tun kann (die Ausführungsrolle) und wer auf Ihre Webfunktion zugreifen kann (ressourcenbasierte Richtlinien). Auf dieser Seite wird erklärt, wie die einzelnen Typen konfiguriert werden.

- Ausführungsrolle — Eine IAM-Rolle, die Ihr Web-Funktionscode für den Zugriff auf AWS Dienste wie Amazon CloudWatch Logs, DynamoDB oder Amazon S3 annimmt.
- Resource-based Richtlinien — Eine mit Ihrer Webfunktion verknüpfte Richtlinie. Es definiert, welche IAM-Prinzipale oder AWS -Dienste Webfunktionsaktionen ausführen können, z. B. das Aufrufen eines Endpunkts, dessen Authentifizierungstyp ist`IamAuth`, was bedeutet, dass Lambda Anfragen mit IAM authentifiziert. Weitere Informationen finden Sie unter the section called “Authentifizierungstyp”. Ein IAM-Prinzipal in einem anderen Konto benötigt ebenfalls identitätsbasierte Berechtigungen.

Das folgende Diagramm zeigt die Identitäten, für die Berechtigungen erforderlich sind, und zeigt, welche Identitäten jeweils verwendet werden. Der Deployer ruft Lambda-Managementaktionen und -anforderungen für die Ausführungsrolle `iam:PassRole` auf. Der Code-Uploader benötigt `s3:PutObject` den versionierten Amazon S3 S3-Bucket, und Lambda liest die Version des Codeobjekts. Lambda kann Ihren Web-Funktionscode und Ihre Umgebungsvariablen mit einem optionalen, vom Kunden verwalteten AWS KMS Schlüssel verschlüsseln. Ihr Code wird mit der Ausführungsrolle ausgeführt, die Zugriff auf Amazon CloudWatch Logs und Ihre Abhängigkeiten gewährt. Ein Endpunkt-Aufrufer benötigt `lambda:InvokeWebFunctionEndpoint` einen `IamAuth` Endpunkt oder Anwendungsanmeldedaten für einen `ApplicationManaged` Endpunkt.



Grant each actor only what its role needs. With IamAuth, endpoint access can also come from a resource-based policy.

Rolle „Ausführung“

Die Ausführungsrolle ist eine IAM-Rolle, die Lambda annimmt, wenn es Ihren Web Function-Code ausführt. Ihre Anwendung verwendet die Berechtigungen dieser Rolle, um mit anderen AWS Diensten zu interagieren. Ihre Webfunktion benötigt mindestens die Erlaubnis, Protokolle in Amazon CloudWatch Logs zu schreiben, damit Sie Ihre Anwendung überwachen können. Wenn Ihre Anwendung aus DynamoDB liest, in Amazon S3 schreibt oder andere AWS Dienste aufruft, fügen Sie diese Berechtigungen der Ausführungsrolle hinzu.

Eine Ausführungsrolle erstellen

Um eine Ausführungsrolle für Ihre Web-Funktion zu erstellen, erstellen Sie eine IAM-Rolle mit einer Vertrauensrichtlinie, die es dem Lambda-Serviceprinzipal ermöglicht, diese zu übernehmen, und fügen Sie eine Berechtigungsrichtlinie hinzu, die den Zugriff gewährt, den Ihre Anwendung benötigt.

Um eine Ausführungsrolle zu erstellen (AWS CLI)

1. Erstellen Sie eine Datei `trust-policy.json` mit dem Namen der folgenden Vertrauensrichtlinie. Diese Richtlinie ermöglicht es dem Lambda-Service Principal, die Rolle in Ihrem Namen zu übernehmen.

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

2. Erstellen Sie die Rolle.

```
aws iam create-role \
    --role-name my-web-function-role \
    --assume-role-policy-document file://trust-policy.json
```

3. Fügen Sie die AWS Lambda grundlegende Richtlinie für die verwaltete Ausführungsrolle hinzu. Diese Richtlinie gewährt Berechtigungen zum Erstellen von Protokollgruppen, zum Protokollieren von Streams und zum Schreiben von Protokollereignissen in Amazon CloudWatch Logs.

```
aws iam attach-role-policy \
    --role-name my-web-function-role \
    --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

Wenn Sie es vorziehen, eine benutzerdefinierte Berechtigungsrichtlinie zu erstellen, anstatt die verwaltete Richtlinie zu verwenden, sind die Mindestberechtigungen, die Ihre Webfunktion für die Protokollierung benötigt:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
```

```
                "arn:aws:logs:us-east-1:111122223333:log-group:/aws/lambda/web/my-web-
app:*",
                "arn:aws:logs:*:111122223333:log-group:/aws/lambda/web/my-web-
app.local:*"
            ]
        }
    ]
}
```

Die Richtlinie deckt zwei Protokollgruppen ab. Lambda schreibt regionsinterne Protokolle in die `.local` Protokollgruppe und zentralisiert sie dann in der Protokollgruppe ohne das Suffix in der Heimatregion. Die zentrale Protokollgruppe ARN verwendet die Heimatregion. Der `.local` ARN verwendet einen Platzhalter „Region“, da Lambda in diese Gruppe in jeder Region schreibt, in der ein Endpunkt Traffic bedient. Auf diese Weise kann eine Ausführungsrolle Endpunkte mit unterschiedlichen Regionsgruppen unterstützen. Wenn die Rolle nicht in die `.local` Protokollgruppe in einer Region schreiben kann, liefert Lambda keine Protokolle aus dieser Region. Wenn Sie eine benutzerdefinierte Protokollgruppe verwenden, ersetzen Sie beide ARNs durch Ihren Protokollgruppennamen und Ihren Protokollgruppennamen, gefolgt von. `.local` Weitere Informationen finden Sie unter the section called “Protokollgruppen”.

Berechtigungen für andere Dienste hinzufügen

Wenn Ihre Anwendung auf andere AWS Dienste zugreift, fügen Sie der Ausführungsrolle die entsprechenden Berechtigungen hinzu. Das folgende Beispiel fügt Lesezugriff auf eine DynamoDB-Tabelle und Schreibzugriff auf einen Amazon S3 S3-Bucket hinzu.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:us-east-1:111122223333:log-group:/aws/lambda/web/my-web-
app:*",
                "arn:aws:logs:*:111122223333:log-group:/aws/lambda/web/my-web-
app.local:*"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:Query"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::my-output-bucket/*"
        }
    ]
}
```

Die IAM-Identität, die eine Webfunktion oder Revision erstellt, verwendet ihre eigenen Berechtigungen, nicht die Ausführungsrolle. Diese Identität benötigt `lambda:CreateWebFunction` oder`lambda:CreateWebFunctionRevision`, `iam:PassRole` in der Ausführungsrolle, Lesezugriff auf das Bereitstellungspaket in Amazon S3. Informationen zu den Amazon S3 S3-Berechtigungen finden Sie unterthe section called “Berechtigungen des Anrufers”. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, um Ihren Web-Funktionscode und Ihre Umgebungsvariablen zu verschlüsseln, benötigt diese Identität ebenfalls AWS KMS Berechtigungen. Weitere Informationen finden Sie unter the section called “Verschlüsselung im Ruhezustand”.

Resource-based Richtlinien

Eine ressourcenbasierte Richtlinie ist ein JSON-Richtliniendokument, das Sie an Ihre Webfunktion anhängen. Sie kann jede Webfunktionsaktion gewähren, mit Ausnahme `lambda:CreateWebFunction` von IAM-Prinzipalen oder -Diensten. AWS Sie kann beispielsweise `lambda:InvokeWebFunctionEndpoint` für einen `IamAuth` Endpunkt gewähren. Es kann auch Verwaltungsaktionen wie `lambda:GetWebFunction`lambda:CreateWebFunctionRevision, und gewährenlambda:UpdateWebFunctionEndpoint.

Der Richtlinienanhang und die zugehörigen Erklärungen verwenden unterschiedliche Ressourcen-ARNs:

- Webfunktions-ARN — Verwenden Sie diesen ARN mit `PutResourcePolicyGetResourcePolicy`,`DeleteResourcePolicy`, und in dem `Resource` Element einer Anweisung, die eine Aktion für die Webfunktion selbst gewährt, z. B. `lambda:GetWebFunctionlambda:CreateWebFunctionRevision`, oder`lambda:CreateWebFunctionEndpoint`. Beispiel, `arn:aws:lambda:us-east-1:111122223333:web-function/my-function`. Verwenden Sie für eine Aktion auf einem vorhandenen Endpunkt`lambda:UpdateWebFunctionEndpoint`, z. B. den Endpunkt-ARN. Verwenden Sie für eine Aktion für eine bestehende Revision`lambda:DeleteWebFunctionRevision`, z. B. den Revision-ARN. Um alle Endpunkte der Web-Funktion abzudecken, verwenden Sie`arn:aws:lambda:us-east-1:111122223333:web-function/my-function/endpoint/*`.
- Endpunkt-ARN — Verwenden Sie diesen ARN als `Resource` Element einer Richtlinienerklärung, die gewährt`lambda:InvokeWebFunctionEndpoint`. Beispiel, `arn:aws:lambda:us-east-1:111122223333:web-function/my-function/endpoint/prod`.

Für einen IAM-Prinzipal in demselben Konto kann entweder die identitätsbasierte Richtlinie des Prinzipals oder die ressourcenbasierte Richtlinie der Webfunktion die Erlaubnis erteilen. Für einen IAM-Prinzipal in einem anderen Konto müssen beide Richtlinien eine Genehmigung gewähren. Dies gilt sowohl für Aufruf- als auch für Verwaltungsaktionen.

Verwenden Sie die attributebasierte Zugriffskontrolle (ABAC), um Aktionen, die auf den Tags einer Webfunktion basieren, zuzulassen oder zu verweigern. Informationen zu den unterstützten Bedingungsschlüsseln finden Sie unter. the section called “Tags (Markierungen)”

> **Note**
>
> Webfunktionen unterstützen`AddPermission`,`RemovePermission`, oder nicht`GetPolicy`. Verwalten Sie die Richtlinie mit `PutResourcePolicyGetResourcePolicy`, und`DeleteResourcePolicy`.

Verwenden Sie ressourcenbasierte Richtlinien, wenn:

- Sie möchten einem Principal in einem anderen Modus Aufrufzugriff gewähren AWS-Konto (kontenübergreifender Aufruf).
- Sie möchten Aufrufzugriff auf einen AWS Dienst gewähren, wie z. CloudFront

- Sie möchten einem Principal in einem anderen die AWS-Konto Verwaltung der Web-Funktion überlassen, um beispielsweise Revisionen zu erstellen oder Endpunkte zu aktualisieren.
- Bei Aufrufen mit demselben Konto können Sie die IAM-Richtlinie des Anrufers nicht ändern (z. B. wenn der Anrufer von einem anderen Team verwaltet wird).

Öffentliche Richtlinien sind nicht zulässig

Sie können einer Webfunktion keine auf öffentlichen Ressourcen basierende Richtlinie zuordnen. Wenn eine Richtlinie öffentlich ist, schlägt die Anfrage mit einem fehl. `ValidationException`

In diesem Abschnitt beschreibt public eine Richtlinie, die Zugriff auf eine nicht spezifizierte Gruppe von Prinzipalen gewährt. Sie beschreibt nicht den Netzwerkzugriff oder den Authentifizierungstyp des Endpunkts.

Was macht eine Richtlinie öffentlich

Eine Richtlinie ist öffentlich, es sei denn, sie beschränkt, wer die Ressource aufrufen kann, auf eine genau spezifizierte Gruppe von Prinzipalen. Lambda geht davon aus, dass die Richtlinie öffentlich ist, und bewertet dann, ob sie als nicht öffentlich gilt. Eine Richtlinie ist öffentlich, wenn sie einen Prinzipal zulässt, von dem jeder AWS Kunde ausgehen könnte. Der häufigste Fall ist ein uneingeschränkter Platzhalterprinzipal, wie im folgenden Beispiel. Lambda lehnt diese Richtlinie mit einem ab. `ValidationException` Benutze es nicht.

```
{
  "Effect": "Allow",
  "Principal": "*",
  "Action": "lambda:InvokeWebFunctionEndpoint",
  "Resource": "arn:aws:lambda:us-west-2:123456789012:web-function/my-web-function/
endpoint/prod"
}
```

Die folgenden Einschränkungen machen eine Richtlinie nicht unveröffentlicht:

- Bedingungen für IP-Adressen. Eine Eingrenzung `aws:SourceIp` auf einen CIDR-Bereich ist nicht ausreichend.
- Föderierte Prinzipale. Ein `Principal` Block, der einen föderierten Identitätsanbieter benennt, schränkt den Zugriff auf eine genau spezifizierte Gruppe von Identitäten nicht ein.

- Bedingungsschlüssel, die die Ressource beschreiben. Schlüssel wie der Authentifizierungstyp des Endpunkts beschreiben, wie die Ressource konfiguriert ist, nicht, wer anruft, sodass sie den Prinzipal nicht einschränken können.

Es gibt zwei Möglichkeiten, eine Richtlinie nicht öffentlich zu machen.

Geben Sie die Principals im Principal-Block an

Listet einen oder mehrere vollständig angegebene Prinzipale im Block auf`Principal`. Die Werte dürfen keine Platzhalter oder Richtlinienvariablen enthalten. Die Anzahl der Principals spielt keine Rolle.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "123456789012",
      "111122223333"
    ]
  },
  "Action": "lambda:InvokeWebFunctionEndpoint",
  "Resource": "arn:aws:lambda:us-west-2:123456789012:web-function/my-web-function/
endpoint/prod"
}
```

Ein IAM-Prinzipal in einem anderen Konto benötigt ebenfalls eine identitätsbasierte Berechtigung, um den Endpunkt aufzurufen.

Verwenden Sie einen vertrauenswürdigen Bedingungsschlüssel

Beschränken Sie einen der folgenden Bedingungsschlüssel auf einen festen Wert. Diese Schlüssel identifizieren den authentifizierten Anrufer und können nicht vom Anforderer festgelegt werden.

- `aws:PrincipalAccount`
- `aws:PrincipalArn`
- `aws:PrincipalOrgID`
- `aws:PrincipalOrgPaths`
- `aws:SourceAccount`
- `aws:SourceArn`

- aws:SourceOrgID
- aws:SourceOrgPaths
- aws:SourceVpc
- aws:SourceVpcArn
- aws:SourceVpce
- aws:VpceAccount
- aws:VpceOrgID
- aws:VpceOrgPaths
- aws:Ec2InstanceSourceVpc
- aws:UserId
- aws:PrincipalServiceName
- aws:PrincipalServiceNamesList
- aws:PrincipalIsAWSService
- lambda:SourceFunctionArn

Die Benennung eines vertrauenswürdigen Schlüssels allein reicht nicht aus. Sie müssen den Wert auch einschränken:

- Zeichenkettenwerte dürfen keine Platzhalter oder Richtlinienvariablen enthalten.
- ARN-Werte dürfen keine Platzhalter oder Richtlinienvariablen in den ersten fünf Feldern des ARN enthalten:arn, der Partition, dem Dienst, der Region und der Konto-ID. Platzhalter im Ressourcenteil sind zulässig.
- muss aws:PrincipalIsAWSService auf true festgelegt sein.

Die folgende Richtlinie gewährt Prinzipalen in Ihrer Organisation die ressourcenbasierte Berechtigung, die zum Aufrufen des Endpunkts erforderlich ist. Sie gilt nicht als öffentlich. Ein IAM-Prinzipal in einem anderen Konto benötigt ebenfalls eine identitätsbasierte Berechtigung.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowOrgAccess",
      "Effect": "Allow",
```

```
      "Principal": {
        "AWS": "*"
      },
      "Action": "lambda:InvokeWebFunctionEndpoint",
      "Resource": "arn:aws:lambda:us-west-2:123456789012:web-function/my-web-function/
endpoint/prod",
      "Condition": {
        "StringEquals": {
          "aws:PrincipalOrgID": "o-a1b2c3d4e5"
        }
      }
    }
  ]
}
```

Cross-account Zugriff

Die folgende ressourcenbasierte Richtlinie gewährt einer bestimmten IAM-Rolle in einem anderen Konto die Berechtigung, einen Endpunkt Ihrer Webfunktion aufzurufen. Die Rolle benötigt außerdem eine identitätsbasierte Berechtigung, um den Endpunkt aufzurufen.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCrossAccountInvoke",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/caller-role"
            },
            "Action": "lambda:InvokeWebFunctionEndpoint",
            "Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/my-
function/endpoint/prod"
        }
    ]
}
```

Damit der kontenübergreifende Aufruf erfolgreich ist, müssen alle der folgenden Bedingungen erfüllt sein:

- Die identitätsbasierte Richtlinie des Aufrufers gewährt `lambda:InvokeWebFunctionEndpoint` Berechtigungen für den Endpunkt-ARN.

- Die ressourcenbasierte Richtlinie in der Webfunktion gewährt dem Aufrufer Berechtigungen (wie oben gezeigt).
- Der Endpunkt verwendet `IamAuth` Authentifizierung, sodass Lambda die Anmeldeinformationen des Anrufers auswertet.

Gewährung des Zugriffs auf ein AWS Service nicht zulässig

Die folgende ressourcenbasierte Richtlinie gewährt dem CloudFront Dienstprinzipal die Erlaubnis, Ihre Webfunktion aufzurufen. Verwenden Sie diese Richtlinie, wenn Sie eine CloudFront Distribution mit Ihrem Web Function-Endpunkt als Ursprung konfigurieren.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCloudFrontInvoke",
            "Effect": "Allow",
            "Principal": {
                "Service": "cloudfront.amazonaws.com"
            },
            "Action": "lambda:InvokeWebFunctionEndpoint",
            "Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/my-
function/endpoint/prod",
            "Condition": {
                "ArnLike": {
                    "aws:SourceArn":
 "arn:aws:cloudfront::111122223333:distribution/EDFDVBD6EXAMPLE"
                }
            }
        }
    ]
}
```

Die `aws:SourceArn` Bedingung schränkt den Zugriff auf eine bestimmte CloudFront Distribution ein. Ohne diese Bedingung könnte jede CloudFront Verteilung in einem beliebigen Konto Ihre Webfunktion aufrufen.

Replikation und letztendliche Konsistenz

Wenn Sie eine ressourcenbasierte Richtlinie an eine Webfunktion anhängen oder aktualisieren, die über PerRegionEndpunkte verfügt MultiRegion, repliziert Lambda die Richtlinie auf alle AWS-

Regionen Replikate, auf denen die Endgeräte betrieben werden. Diese Replikation ist letztendlich konsistent.

Während des Replikationsfensters bewertet Lambda möglicherweise Anfragen in Replikatregionen anhand der vorherigen Version der Richtlinie. Das bedeutet Folgendes:

- Eine neu hinzugefügte Genehmigung wird möglicherweise nicht sofort in allen Regionen wirksam.
- Lambda respektiert in einigen Regionen möglicherweise immer noch eine kürzlich entfernte Berechtigung, bis die Replikation abgeschlossen ist.

Es gibt kein für den Kunden sichtbares Abschlusssignal für die Richtlinienreplikation. Testen Sie das erwartete Zulassen- oder Ablehnungsverhalten in jeder Endpunktregion, bevor Sie sich auf die neue Richtlinie verlassen. `throttleConfig.rateLimit`Um Notfälle einzudämmen, aktualisieren Sie das Routing des Datenverkehrs oder stellen Sie die Endgeräte so ein, `0` dass der gesamte Datenverkehr blockiert wird, solange sich die Richtlinienänderung verbreitet. Weitere Informationen finden Sie unter the section called “Drosselung”.

Für HomeRegionEndgeräte gilt die Richtlinie nur in der Heimatregion und erfordert keine regionsübergreifende Replikation. Es kann immer noch eine Weile dauern, bis eine Änderung wirksam wird.

Berechtigungen für die Netzwerkkonfiguration

Die Identität, die das Netzwerk konfiguriert, benötigt die Erlaubnis, die entsprechenden Lambda-Ressourcen zu erstellen oder zu aktualisieren. Außerdem benötigt sie die Erlaubnis, die erforderlichen Rollen und den Konnektor zu übergeben. Je nach Vorgang umfassen die Berechtigungen:

- `lambda:CreateNetworkConnector`
- `lambda:UpdateNetworkConnector`
- `lambda:GetNetworkConnector`
- `lambda:CreateWebFunction`
- `lambda:CreateWebFunctionRevision`
- `lambda:UpdateWebFunctionEndpoint`
- `lambda:PassNetworkConnector`
- `iam:PassRole`

Der Anrufer muss `lambda:PassNetworkConnector` auf dem Connector, auf den verwiesen wird, immer wenn ein `create-web-function` `create-web-function-revision` Oder-Aufruf gesetzt `egressNetworkConnectorArn` wird, aktiviert sein. Schränken Sie diese Berechtigung auf genehmigte Connector-Ressourcen ein. Der Geltungsbereich erstreckt `iam:PassRole` sich auf genehmigte Ausführungsrollen, die Sie beim Erstellen einer Webfunktion oder -version übergeben, und auf genehmigte Operatorrollen, die Sie beim Erstellen oder Aktualisieren eines Netzwerkconnectors übergeben. Informationen zur Richtlinie über Berechtigungen für Operatorrollen finden Sie unterthe section called “Berechtigungen für die Operatorrolle”.

Endpunktzugriffskontrolle mit Lambda Web Function

Sie können den Zugriff auf Ihre AWS Lambda Web Function-Endpunkte mithilfe der Einstellungen steuern, die `AuthType` Sie für jeden Endpunkt festgelegt haben. Je nach Authentifizierungstyp können Sie der Web-Funktion auch eine ressourcenbasierte Richtlinie hinzufügen, um zu steuern, wer einen Endpunkt aufrufen kann.

Wenn Sie einen Endpunkt konfigurieren, geben Sie eine der folgenden Optionen an: `AuthType`

`IamAuth`

Bei diesem Authentifizierungstyp verwendet Lambda AWS Identity and Access Management (IAM), um Anfragen auf der Grundlage der Identitätsrichtlinie des Anrufers und der ressourcenbasierten Richtlinie der Webfunktion zu authentifizieren und zu autorisieren. Entweder signiert der Anrufer Anfragen direkt mit AWS Signature Version 4 (Sigv4) oder mit Signature Version 4a (SigV4a) für `MultiRegion` Endgeräte, oder Amazon CloudFront signiert in seinem Namen, wenn er den Endpunkt öffnet (siehe). AWS-Service the section called “CloudFront Integration” Wählen Sie diese Option, wenn Lambda die IAM-based Zugriffskontrolle auf Ihrem Endpunkt durchsetzen soll.

`ApplicationManaged`

Lambda authentifiziert oder autorisiert keine Anfragen. Jede Anfrage erreicht Ihre Funktion, und Ihr Funktionscode ist für jede Authentifizierung verantwortlich. Wählen Sie diese Option für öffentliche Endpunkte oder wenn Sie die Authentifizierung in Ihrem eigenen Code durchführen.

Die IAM-Aktion zum Aufrufen eines Endpunkts ist. `lambda:InvokeWebFunctionEndpoint`

Sie legen den Authentifizierungstyp eines Endpunkts mit dem `--auth-type` Parameter fest, wenn Sie den Endpunkt (`create-web-function-endpoint`) erstellen, und Sie können ihn später mit

ändern. `update-web-function-endpoint` Informationen zum Befehl create finden Sie unterthe section called “Erstellen eines Endpunkts”. Informationen zu den Eigenschaften, die Sie ändern können, finden Sie unterthe section called “Aktualisierung eines Endpunkts”.

Verwenden Sie den **`IamAuth`**Authentifizierungstyp

Wenn Sie den `IamAuth` Authentifizierungstyp wählen, müssen Anrufer jede Anfrage mit SigV4 (SigV4a für `MultiRegion` Endpunkte) signieren und dürfen den Endpunkt aufrufen. Je nachdem, wer die Anfrage stellt, erteilen Sie diese Berechtigung mit einer ressourcenbasierten Richtlinie für die Webfunktion, einer identitätsbasierten Richtlinie für den Anrufer oder mit beidem.

Befindet sich der Anrufer in derselben AWS-Konto Webfunktion, kann er entweder `lambda:InvokeWebFunctionEndpoint` über eine identitätsbasierte Richtlinie oder über die ressourcenbasierte Richtlinie der Funktion erteilt werden. Wenn der Anrufer ein anderes Konto hat, benötigt er sowohl eine identitätsbasierte Richtlinie, die die Aktion gewährt, als auch eine ressourcenbasierte Richtlinie für die Funktion, die dies zulässt.

Resource-based Richtlinien sind an die Webfunktion angehängt, aber jede Anweisung kann sich `Resource` auf einen bestimmten Endpunkt-ARN ARN. Die folgende ressourcenbasierte Richtlinie ermöglicht es der `InvokeRole` Rolle im Konto`222233334444`, den Endpunkt von aufzurufen: `prod` `my-function`

Example Cross-account ressourcenbasierte Richtlinie

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::222233334444:role/InvokeRole"
            },
            "Action": "lambda:InvokeWebFunctionEndpoint",
            "Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/my-
function/endpoint/prod"
        }
    ]
}
```

Verwenden `put-resource-policy` Sie zum Anhängen der ressourcenbasierten Richtlinie die Funktion ARN (kein Endpunkt-ARN):

```
aws lambda-web put-resource-policy \
    --resource-arn "arn:aws:lambda:us-east-1:111122223333:web-function/my-function" \
    --policy file://policy.json \
    --region us-east-1
```

Sie können die Richtlinie mit `get-resource-policy` und überprüfen oder entfernen. `delete-resource-policy`

Eine Webfunktion hat eine ressourcenbasierte Richtlinie und `put-resource-policy` ersetzt diese. Um eine Anweisung hinzuzufügen, z. B. die CloudFront Anweisung inthe section called “CloudFront Integration”, rufen Sie die aktuelle Richtlinie mit ab`get-resource-policy`, fügen die Anweisung dem zugehörigen `Statement` Array hinzu und übergeben das zurückgegebene `revisionId` mit. `--revision-id` Wenn jemand die Richtlinie geändert hat, nachdem Sie sie gelesen haben, schlägt die Anfrage fehl und `ConflictException` die Änderung wird nicht überschrieben.

Um das Aufrufen aller Endpunkte einer Funktion zu ermöglichen, verwenden Sie einen Platzhalter im Ressourcen-ARN:

```
"Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/my-function/endpoint/*"
```

Ersetzen Sie die Region, die Konto-ID und den Funktionsnamen durch Ihre eigenen Werte. Bleib `endpoint/*` am Ende.

Einem Principal die Erlaubnis zum Aufrufen erteilen

Ein Aufrufer `lambda:InvokeWebFunctionEndpoint` muss einen Endpunkt aufrufen. Für einen Anrufer mit demselben Konto wie die Web-Funktion ist es ausreichend, dem Anrufer diese Aktion in der identitätsbasierten Richtlinie zuzuweisen. Es ist keine ressourcenbasierte Richtlinie erforderlich. Für einen Anrufer mit einem anderen Konto ist diese identitätsbasierte Richtlinie zusätzlich zu einer ressourcenbasierten Richtlinie für die Funktion erforderlich, die dies zulässt.

Fügen Sie der IAM-Rolle oder dem IAM-Benutzer, der den Endpunkt aufruft, die folgende Richtlinie hinzu:

Example Identity-based Richtlinie zum Aufrufen eines Endpunkts

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": "lambda:InvokeWebFunctionEndpoint",
            "Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/my-
function/endpoint/prod"
        }
    ]
}
```

Signieren von Anforderungen

Lambda bietet kein spezielles SDK zum Aufrufen von Endpunkten. Sie senden Standard-HTTP-Anfragen und signieren sie selbst mit SigV4 oder SigV4a, wobei Sie den Dienstnamen verwenden. `lambda` Beachten Sie Folgendes:

- Jede signierte Anfrage muss den `X-Amz-Content-SHA256` Header enthalten, auch Anfragen ohne Hauptteil. Lambda liest den Anforderungstext nicht, wenn es die Signatur validiert, und verwendet daher diesen Header-Wert als Nutzlast-Hash. Anfragen ohne diesen Wert werden mit abgelehnt. `403 Forbidden`
- Wenn Sie den Text nicht signieren (z. B. beim Streamen), verwenden Sie den Literalwert `UNSIGNED-PAYLOAD` für`X-Amz-Content-SHA256`. Lambda unterstützt kein Chunked Body Signing.
- Lambda vergleicht den Anforderungstext nicht mit dem `X-Amz-Content-SHA256` Wert. Die Signatur deckt den Header-Wert ab, nicht den Hauptinhalt. Wenn Sie einen Schutz der Körperintegrität benötigen, der über TLS hinausgeht, implementieren Sie ihn in Ihrer Anwendung.
- Regionale Endpunkte (`HomeRegion`,`PerRegion`) verwenden SigV4 mit der Region des Endpunkts. `MultiRegion`Endpunkte verwenden AWS Signature Version 4a (SigV4a), die für eine Reihe von Regionen statt für eine Region signiert. Verwenden Sie das Regions-Set. *
- Wenn eine `IamAuth` Anfrage zurückkehrt`403 Forbidden`, überprüfen Sie, ob die Anfrage mit SigV4 (oder SigV4a) signiert ist, den `X-Amz-Content-SHA256` Header enthält und nicht abgelaufene Anmeldeinformationen verwendet und ob Ihre Richtlinien `lambda:InvokeWebFunctionEndpoint` auf dem Endpunkt-ARN dies zulassen.

Weitere Informationen zum Signieren von Anfragen finden Sie unter Signieren von API-Anfragen. AWS

Nachdem Lambda die Sigv4- oder SigV4A-Signatur validiert hat, entfernt es den `Authorization` Header, bevor die Anfrage an Ihre Funktion weitergeleitet wird. Wenn Sie möchten, dass Ihre Funktion den `Authorization` Header auf einem `IamAuth` Endpunkt empfängt, senden Sie

Anfragen über CloudFront, der sie in Ihrem Namen signiert (siehe). the section called “CloudFront Integration”

CloudFront Integration

Sie können Amazon CloudFront Origin Access Control (OAC) verwenden, um eine CloudFront Distribution vor einen `IamAuth` Endpunkt zu stellen. Bei OAC CloudFront signiert jede ursprüngliche Anfrage im Namen des Betrachters, sodass Anrufer Anfragen nicht selbst unterschreiben. CloudFront verwendet den `Authorization` Header nicht, um die ursprüngliche Anfrage zu signieren, sodass es Ihren `Authorization` Header an die Funktion weiterleiten kann.

Lambda entfernt jeden `Authorization` Wert, der selbst eine AWS Signatur ist (ein Wert, der mit beginnt`AWS4-`). Auf diese Weise können Sie die Authentifizierung Ihrer Anwendung (z. B. JWT oder Cookies) beibehalten und gleichzeitig jede ursprüngliche CloudFront Anfrage signieren. Konfigurieren Sie dazu die Distribution so, dass sie den `Authorization` Header weiterleitet. Weitere Informationen finden Sie unter Beschränken des Zugriffs auf einen AWS Ursprung.

Die Endpunktdomäne bleibt im Internet erreichbar, und Lambda akzeptiert auch direkt signierte Anfragen von jedem Prinzipal, den Ihre Richtlinien zulassen.

Dies setzt voraus, dass der Authentifizierungstyp des Endpunkts `IamAuth` Erstellen Sie die Origin-Zugriffskontrolle mit Ursprungstyp `lambda` und Signaturverhalten`always-amz-auth`. CloudFront lehnt andere Signaturverhalten für Webfunktions-Ursprünge ab.

> **Note**
>
> Lambda authentifiziert keine Anfragen an einen `ApplicationManaged` Endpunkt und bewertet auch nicht die ressourcenbasierte Richtlinie der Webfunktion für sie. Ein solcher Endpunkt bleibt unter seiner eigenen Domain öffentlich erreichbar, auch wenn er dahinter liegt. CloudFront

Um den Endpunkt aufrufen CloudFront zu können, fügen Sie der Webfunktion eine ressourcenbasierte Richtlinie hinzu, die dem CloudFront Dienstprinzipal (`cloudfront.amazonaws.com`) die Aktion erteilt. `lambda:InvokeWebFunctionEndpoint` Verwenden Sie eine `aws:SourceArn` Bedingung, um nur Ihre spezifische Distribution zuzulassen:

Example Resource-based Richtlinie für CloudFront OAC

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCloudFrontOACAccess",
            "Effect": "Allow",
            "Principal": {
                "Service": "cloudfront.amazonaws.com"
            },
            "Action": "lambda:InvokeWebFunctionEndpoint",
            "Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/my-
function/endpoint/prod",
            "Condition": {
                "StringEquals": {
                    "aws:SourceArn": "arn:aws:cloudfront::111122223333:distribution/
<distribution-id>"
                }
            }
        }
    ]
}
```

Verwenden Sie den **`ApplicationManaged`**Authentifizierungstyp

Wenn der Authentifizierungstyp eines Endpunkts lautet`ApplicationManaged`, kann jeder, der die Domäne des Endpunkts besitzt, Ihre Funktion aufrufen. Lambda führt keine Authentifizierung oder Autorisierung durch.

Wählen Sie aus, `ApplicationManaged` wann Ihr Endpunkt öffentlich erreichbar sein soll, um beispielsweise Anfragen direkt von einem Webbrowser aus zu bearbeiten. Sie können es auch wählen, wenn Sie die Authentifizierung in Ihrem eigenen Code abwickeln, z. B. OAuth 2.0, OpenID Connect oder einem benutzerdefinierten Token-Schema.

Mit diesem Authentifizierungstyp:

- Lambda ermöglicht es allen Aufrufern, den Endpunkt aufzurufen. Es bewertet keine ressourcenbasierte Richtlinie für diese Anfragen.
- Lambda leitet den `Authorization` Header des Aufrufers an Ihre Funktion weiter, sodass Ihr Code ihn validieren kann. Wenn der Header eine AWS Signature Version 4 (Sigv4) -Signatur enthält, entfernt Lambda sie zusammen mit allen `X-Amz-Security-Token` Headern, sodass AWS Anmeldeinformationen nicht an Ihren Funktionscode übergeben werden.
- Ihre Funktion ist für die gesamte Authentifizierung und Autorisierung verantwortlich.

> **Note**
>
> Bei `ApplicationManaged` Endpunkten bewertet Lambda die ressourcenbasierte Richtlinie der Webfunktion für Aufrufanforderungen nicht. Sie können eine ressourcenbasierte Richtlinie nicht verwenden, um einzuschränken, wer einen Endpunkt aufruft. `ApplicationManaged` Um den Aufruf einzuschränken, verwenden Sie stattdessen. `IamAuth` Die ressourcenbasierte Richtlinie gilt weiterhin für Verwaltungsaktionen. Beispielsweise kann sie einem anderen Konto die Erlaubnis erteilen, den Endpunkt zu aktualisieren. Weitere Informationen finden Sie unter the section called “Resource-based Richtlinien”.

Governance und Zugriffskontrolle

Sie können steuern, wer Endpunkte konfigurieren kann.

Lambda unterstützt die folgenden IAM-Aktionen für die Verwaltung von Endpunkten:

- `lambda:CreateWebFunction`— Erstellen Sie eine Webfunktion und konfigurieren Sie ihren anfänglichen Endpunkt, einschließlich ihres. `AuthType`
- `lambda:CreateWebFunctionEndpoint`— Erstellen Sie einen Endpunkt und legen Sie seinen fest`AuthType`.
- `lambda:UpdateWebFunctionEndpoint`— Aktualisiere einen Endpunkt, einschließlich seines`AuthType`.
- `lambda:GetWebFunctionEndpoint`— Zeigt die Konfiguration eines Endpunkts an.
- `lambda:ListWebFunctionEndpoints`— Listet die Endpunkte einer Webfunktion auf.
- `lambda:DeleteWebFunctionEndpoint`— Löscht einen Endpunkt.

Verwenden Sie einen Bedingungsschlüssel, um eine Aktion auf der Grundlage eines Authentifizierungstyps zuzulassen oder abzulehnen. `lambda:Request/WebFunctionAuthType`gilt für `CreateWebFunction``CreateWebFunctionEndpoint`, und`UpdateWebFunctionEndpoint`. `lambda:Resource/WebFunctionAuthType`gilt für `InvokeWebFunctionEndpoint` (auf `IamAuth` Endpunkten), `UpdateWebFunctionEndpoint``GetWebFunctionEndpoint`, und`DeleteWebFunctionEndpoint`. Details hierzu finden Sie unter the section called “Bedingungsschlüssel”.

Einschränkung öffentlicher Endpunkte in einer Organisation

Wenn Sie eine AWS Organisation verwalten und Entwickler daran hindern möchten, öffentliche Endpunkte zu erstellen, verwenden Sie eine Service Control Policy (SCP), die die Erstellung oder Aktualisierung von Endpunkten mit dem Authentifizierungstyp verweigert. `ApplicationManaged` Verwenden Sie den `lambda:Request/WebFunctionAuthType` Bedingungsschlüssel, der den Authentifizierungstyp in der Anfrage widerspiegelt:

Example SCP verweigert öffentliche Endpunkte

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "lambda:CreateWebFunction",
                "lambda:CreateWebFunctionEndpoint",
                "lambda:UpdateWebFunctionEndpoint"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:Request/WebFunctionAuthType": "ApplicationManaged"
                }
            }
        }
    ]
}
```

Die Richtlinie schließt ein, `lambda:CreateWebFunction` weil bei diesem Vorgang der erste Endpunkt anhand des Endpunkts der Anfrage erstellt wird. `endpointConfig` Wenn diese Richtlinie aktiviert ist, `aws lambda-web deploy` schlägt der Fehler mit der Meldung „Zugriff verweigert“ fehl, sofern der Entwickler nichts hinzufügt`--auth-type IamAuth`. Standardmäßig erstellt der Befehl einen `ApplicationManaged` Endpunkt.

Bedingungsschlüssel

Für eine differenzierte Zugriffskontrolle unterstützt Lambda die folgenden Bedingungskontextschlüssel:

- `lambda:Request/WebFunctionAuthType`— Der in einer Anfrage angegebene Authentifizierungstyp. Verfügbar während `CreateWebFunctionCreateWebFunctionEndpoint`, und`UpdateWebFunctionEndpoint`. Verwenden Sie es, um zu steuern, welche Authentifizierungstypen konfiguriert werden können.
- `lambda:Resource/WebFunctionAuthType`— Der aktuell auf einem Endpunkt konfigurierte Authentifizierungstyp mit den Werten `IamAuth` und`ApplicationManaged`. Verfügbar während `InvokeWebFunctionEndpointUpdateWebFunctionEndpoint`,`DeleteWebFunctionEndpoint`, und`GetWebFunctionEndpoint`. Denn `InvokeWebFunctionEndpoint` Lambda wertet den Schlüssel nur auf `IamAuth` Endpunkten aus, da Anfragen an `ApplicationManaged` Endpunkte nicht mit IAM autorisiert sind.
- `aws:ResourceTag/${TagKey}`— Filtert den Zugriff anhand der Tags in der Webfunktion. Endpunkte und Revisionen erben die Tags der übergeordneten Funktion, sodass Sie ABAC verwenden können, um den Aufrufzugriff zu kontrollieren.

Example Erlauben Sie den Aufruf nur auf Endpunkten einer markierten Funktion

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeWebFunctionEndpoint",
            "Resource": "arn:aws:lambda:us-east-1:111122223333:web-function/*/endpoint/
prod",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Environment": "Production"
                }
            }
        }
    ]
}
```

Verschlüsselung im Ruhezustand für Webfunktionen

AWS Lambda verschlüsselt die Daten für Ihre Lambda Web Functions im Ruhezustand auf der Serverseite. Auf dieser Seite werden die verfügbaren Verschlüsselungsoptionen und die Konfiguration eines vom Kunden verwalteten Schlüssels erläutert.

Optionen für die Verschlüsselung im Ruhezustand

Die Verschlüsselung im Ruhezustand ist immer aktiviert. Lambda verschlüsselt Ihr Webfunktions-Code-Artefakt und Ihre Umgebungsvariablen auf der Serverseite, und Sie können diese Verschlüsselung nicht ausschalten. Standardmäßig verwendet Lambda eine AWS-eigener Schlüssel. Sie können stattdessen einen vom Kunden verwalteten Schlüssel konfigurieren.

Sie haben zwei Optionen für den AWS KMS Schlüssel, der Ihre Daten schützt:

AWS-eigener Schlüssel (Standard)

Standardmäßig verschlüsselt Lambda Ihre Daten mit einem. AWS-eigener Schlüssel AWS besitzt und verwaltet diesen Schlüssel. Sie können seine Verwendung nicht einsehen, verwalten oder überwachen. Wenn dieses Standardverhalten Ihren Anforderungen entspricht, ist keine zusätzliche Einrichtung erforderlich.

Kundenverwalteter Schlüssel

Sie können optional Ihren eigenen vom AWS KMS Kunden verwalteten Schlüssel konfigurieren. Verwenden Sie einen vom Kunden verwalteten Schlüssel, wenn Sie die Schlüsselrotation kontrollieren, die Schlüsselnutzung überprüfen CloudTrail, den Schlüsselzugriff mithilfe einer wichtigen Richtlinie verwalten oder die Compliance-Anforderungen des Unternehmens erfüllen müssen. Mit einem vom Kunden verwalteten Schlüssel können nur von Ihnen autorisierte Benutzer auf den verschlüsselten Code und die Umgebungsvariablen zugreifen. Für vom Kunden verwaltete Schlüssel fallen Gebühren AWS KMS an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Verschlüsselung von Daten im Ruhezustand mithilfe eines vom Kunden verwalteten Schlüssels

In den folgenden Abschnitten wird erklärt, wie Web Functions einen vom Kunden verwalteten Schlüssel verwenden, wie der Schlüssel und seine Berechtigungen konfiguriert werden und wie der Zugriff darauf eingeschränkt werden kann.

Wie Web Functions einen vom Kunden verwalteten Schlüssel verwendet

Wenn Sie einen vom Kunden verwalteten Schlüssel angeben, verwendet Lambda die Umschlagverschlüsselung. Das bedeutet, dass Lambda die Daten der Revision mit einem

Datenschlüssel verschlüsselt und Ihr KMS-Schlüssel diesen Datenschlüssel verschlüsselt. Lambda ruft den Datenschlüssel von Ihrem KMS-Schlüssel ab AWS KMS und entschlüsselt ihn mit diesem, wenn es die Daten lesen muss. Nur Prinzipale mit Zugriff auf den KMS-Schlüssel können auf den Code und die Umgebungsvariablen der Revision zugreifen.

Lambda verschlüsselt die folgenden Web Function-Ressourcen mit Ihrem vom Kunden verwalteten Schlüssel:

Ressource	Was ist verschlüsselt
Revision	Das aus Ihrem Amazon S3 S3-Bereitstellungspaket erstellte Codeartefakt und die Umgebungsvariablen der Revision.

Lambda verwendet keine AWS KMS Zuschüsse für Webfunktionen. Der Zugriff wird über die Schlüsselrichtlinie und die IAM-Berechtigungen des aufrufenden Prinzipals autorisiert.

Lambda ruft AWS KMS beim Erstellen und Starten von Revisionsressourcen auf und kann entschlüsseltes Material in aktiven Ausführungsumgebungen zwischenspeichern. Das Widerrufen des Schlüsselzugriffs ist kein sofortiger Mechanismus zur Eindämmung des Datenverkehrs, da bestehende Umgebungen zwischengespeichertes Material für einen unbestimmten Zeitraum weiterhin verwenden können.

Testen Sie nach einer Änderung der Schlüsselrichtlinie das erwartete Verhalten und überwachen Sie die Version und den Zustand der Endgeräte. Um den Notfall einzudämmen, beenden Sie die Weiterleitung des Datenverkehrs zu den betroffenen Versionen oder deaktivieren Sie den Endpunkt, anstatt sich nur auf die Sperrung von Schlüsseln zu verlassen.

Lambda legt den folgenden Verschlüsselungskontext für alle AWS KMS Operationen für Webfunktionen fest:

- `aws:lambda:WebFunctionRevisionArn`— Der Amazon-Ressourcenname (ARN) der Web Function-Revision.

Sie können diesen Verschlüsselungskontext in wichtigen Richtlinien verwenden, um den Zugriff auf bestimmte Webfunktionen oder Revisionen einzuschränken. Weitere Informationen finden Sie unter the section called “Einschränkung des Zugriffs auf Ihren KMS-Schlüssel”.

Konfiguration eines vom Kunden verwalteten Schlüssels

Der AWS KMS Schlüssel muss ein symmetrischer Verschlüsselungsschlüssel sein (Schlüsselspezifikation`SYMMETRIC_DEFAULT`, Schlüsselverwendung`ENCRYPT_DECRYPT`). Sie können keine asymmetrischen Schlüssel verwenden.

Webfunktionen unterstützen Tasten für mehrere Regionen. Die Art des Schlüssels, den Sie benötigen, hängt von Ihrer Endpunktkonfiguration ab:

- HomeRegion Endpunkt — Sie können einen Schlüssel für eine Region oder einen Schlüssel für mehrere Regionen verwenden.
- MultiRegion oder PerRegion Endpunkt — Sie müssen einen Schlüssel für mehrere Regionen mit Replikaten in jedem, den der Endpunkt bedient AWS-Region , verwenden. Fehlt in einer Region ein Replikat des Schlüssels, schlägt die Replikation in diese Region fehl.

Informationen zum Erstellen von Schlüsseln für mehrere Regionen finden Sie unter Erstellen von Primärschlüsseln für mehrere Regionen im AWS Key Management Service Entwicklerhandbuch.

Da Schlüsselrichtlinien AWS KMS nicht zwischen verwandten Schlüsseln mit mehreren Regionen kopiert oder synchronisiert werden, müssen Sie jede Schlüsselrichtlinie separat konfigurieren. Konfigurieren Sie die Schlüsselrichtlinie für den Primärschlüssel und für jeden Replikatschlüssel. Die Richtlinie jedes Replikatschlüssels muss mindestens den Aufruf durch den Lambda-Serviceprinzipal ermöglichen. `kms:Decrypt` Diese Berechtigung ermöglicht Lambda, den Datenschlüssel in dieser Region zu entschlüsseln. Weitere Informationen finden Sie unter Autorisieren der Verwendung von Schlüsseln für mehrere Regionen im Entwicklerhandbuch.AWS Key Management Service

> **Note**
>
> Wenn Sie Replikatschlüssel für mehrere Regionen erstellen, kann es einige Sekunden bis einige Minuten dauern, bis sie in allen Regionen verfügbar sind. Vergewissern Sie sich, dass der Replikatschlüssel in jeder Zielregion verfügbar ist, bevor Sie die Revision oder den Endpunkt erstellen.

Konfiguration von Berechtigungen zur Verwendung eines vom Kunden verwalteten Schlüssels

Um eine Revision mit einem vom Kunden verwalteten Schlüssel zu erstellen, benötigen Sie die folgenden Berechtigungen für den AWS KMS Schlüssel:

- `kms:DescribeKey`— Erforderlich, wenn Sie eine Webfunktion oder Revision mit dem Schlüssel erstellen.
- `kms:GenerateDataKey`— Lambda ruft diese Operation auf, um den Datenschlüssel zu generieren, der das Codeartefakt und die Umgebungsvariablen der Revision verschlüsselt.
- `kms:Decrypt`— Lambda ruft diese Operation auf, um zu überprüfen, ob der Schlüssel verwendet werden kann, wenn Sie die Revision erstellen. Prinzipale, die eine Revision lesen, benötigen `GetWebFunctionRevision` auch `kms:Decrypt` den Schlüssel, um die Umgebungsvariablen zu sehen.

Die Erstellungsanforderung schlägt fehl, wenn der Schlüssel fehlt, deaktiviert ist, kein symmetrischer Verschlüsselungsschlüssel ist oder der Zugriff mit den erforderlichen Berechtigungen nicht möglich ist.

> **Note**
>
> Sie benötigen `kms:Encrypt` keine Genehmigung. Lambda verwendet die Envelope-Verschlüsselung mit`GenerateDataKey`, die sowohl eine Klartext- als auch eine verschlüsselte Kopie des Datenschlüssels zurückgibt, ohne dass ein separater Verschlüsselungsaufruf erforderlich ist.

Die AWS KMS Schlüsselrichtlinie muss auch dem Lambda-Service Principal (`lambda.amazonaws.com`) den Aufruf `kms:Decrypt` ermöglichen. Lambda verwendet diese Berechtigung, um den Datenschlüssel zu entschlüsseln, wenn es Ausführungsumgebungen für die Revision startet. Der Service Principal benötigt nicht. `kms:DescribeKey` Der Principal, der die Webfunktion oder Revision erstellt, benötigt diese Berechtigung.

Beispiel für eine Schlüsselrichtlinie

Die folgende wichtige Richtlinie enthält die drei Anweisungen, die für Webfunktionen erforderlich sind, wobei jeweils die unter beschriebenen Bedingungen gelten. the section called “Einschränkung des Zugriffs auf Ihren KMS-Schlüssel” Die erste Anweisung gewährt dem Lambda-Serviceprinzipal

die Berechtigungen, die er benötigt, um Ausführungsumgebungen zu starten. Die zweite und dritte gewähren dem Prinzipal, der Revisionen erstellt, die Berechtigungen, die er zum Zeitpunkt der Erstellung benötigt.

`kms:DescribeKey`befindet sich in einer separaten Anweisung, da es sich um einen Metadatenvorgang handelt, der keinen Verschlüsselungskontext enthält. Eine Bedingung für den Verschlüsselungskontext würde niemals mit einer `DescribeKey` Anfrage übereinstimmen, sodass diese Bedingung nur für die Anweisung gilt, die `kms:GenerateDataKey` und gewährt`kms:Decrypt`.

Dieses Beispiel zeigt nur die Anweisungen, die Lambda für Web Functions benötigt. Fügen Sie diese Anweisungen zu Ihrer bestehenden Schlüsselrichtlinie hinzu, anstatt sie zu ersetzen, und behalten Sie die Anweisungen bei, die Ihren Administratoren Zugriff auf die Verwaltung des Schlüssels gewähren.

Example Wichtige Richtlinie für die Verschlüsselung von Webfunktionen

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Allow Lambda to decrypt the data key for Web Function revisions",
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "kms:Decrypt",
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "kms:EncryptionContext:aws:lambda:WebFunctionRevisionArn":
 "arn:aws:lambda:us-east-1:111122223333:web-function/my-function/revision/*",
                    "aws:SourceArn": "arn:aws:lambda:us-east-1:111122223333:web-
function/my-function/revision/*"
                }
            }
        },
        {
            "Sid": "Allow the revision creator to generate and validate data keys",
            "Effect": "Allow",
            "Principal": { "AWS":
 "arn:aws:iam::111122223333:role/MyWebFunctionDeployRole" },
            "Action": [
                "kms:GenerateDataKey",
```

```
                "kms:Decrypt"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "kms:EncryptionContext:aws:lambda:WebFunctionRevisionArn":
 "arn:aws:lambda:us-east-1:111122223333:web-function/my-function/revision/*"
                },
                "StringEquals": {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        },
        {
            "Sid": "Allow the revision creator to describe the key",
            "Effect": "Allow",
            "Principal": { "AWS":
 "arn:aws:iam::111122223333:role/MyWebFunctionDeployRole" },
            "Action": "kms:DescribeKey",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        }
    ]
}
```

Weitere Informationen zu AWS KMS wichtigen Richtlinien finden Sie unter Wichtige Richtlinien AWS KMS im AWS Key Management Service Entwicklerhandbuch.

Eine Version der Webfunktion mit einem vom Kunden verwalteten Schlüssel erstellen

Um eine Revision mit einem vom Kunden verwalteten Schlüssel zu verschlüsseln, geben Sie den Schlüssel-ARN im `kmsKeyArn` Parameter der Revisionskonfiguration an. Sie können diesen Parameter festlegen, wenn Sie CreateWebFunctionoder CreateWebFunctionRevisionaufrufen. Wenn Sie es weglassen`kmsKeyArn`, verschlüsselt Lambda die Revision mit dem. AWS-eigener Schlüssel

Um eine Webfunktion mit einem vom Kunden verwalteten Schlüssel zu erstellen (AWS CLI)

- Führen Sie den folgenden Befehl aus, um eine Webfunktion mit einem vom Kunden verwalteten Schlüssel zu erstellen.

```
aws lambda-web create-web-function \
    --function-name my-web-app \
    --revision-config '{
      "kmsKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id",
      "buildConfig": {
        "codeConfig": {
          "s3Object": { "bucket": "amzn-s3-demo-bucket", "key": "my-app.zip" }
        },
        "runtimeConfig": { "runtime": "nodejs24.x" }
      },
      "serviceConfig": {
        "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role"
      }
    }' \
    --endpoint-config '{
      "endpointName": "my-endpoint",
      "endpointType": "HomeRegion",
      "authType": "ApplicationManaged"
    }'
```

Ändern der Verschlüsselungskonfiguration einer vorhandenen Webfunktion

Der `kmsKeyArn` Parameter ist für die gesamte Lebensdauer einer Revision festgelegt. Da Revisionen unveränderlich sind, können Sie den Schlüssel, der eine bestehende Revision schützt, nicht ändern. Um die Verschlüsselungskonfiguration zu ändern, erstellen Sie eine neue Version, die den neuen Schlüssel spezifiziert, und leiten Sie dann den Datenverkehr an diese Version weiter.

Webfunktionen unterstützen die folgenden Tastenübergänge:

- AWS-eigener Schlüssel zum vom Kunden verwalteten Schlüssel — Erstellen Sie eine neue Version, die Folgendes festlegt`kmsKeyArn`.
- Ein vom Kunden verwalteter Schlüssel zu einem anderen vom Kunden verwalteten Schlüssel — Erstellen Sie eine neue Version, die den neuen Schlüssel-ARN spezifiziert.
- Vom Kunden verwalteter Schlüssel für AWS-eigener Schlüssel — Erstellen Sie eine neue Version ohne`kmsKeyArn`. Lambda schützt diese Revision dann mit der AWS-eigener Schlüssel, was die Standardeinstellung ist.

Lambda verschlüsselt bestehende Revisionen nicht erneut, wenn Sie eine neue Revision mit einem anderen Schlüssel erstellen. Jede Revision bleibt mit dem Schlüssel verschlüsselt, den Sie bei der Erstellung angegeben haben. Das bedeutet, dass, solange eine frühere Version noch Traffic empfangen kann, ihr ursprünglicher Schlüssel aktiviert und für den Lambda-Serviceprinzipal zugänglich bleiben muss. Bevor Sie einen Schlüssel deaktivieren oder löschen, stellen Sie sicher, dass die Revisionsgewichte eines Endpunkts immer noch auf keine mit diesem Schlüssel verschlüsselte Version verweisen.

Um eine neue Version mit einem anderen vom Kunden verwalteten Schlüssel zu erstellen (AWS CLI)

1. Führen Sie den folgenden Befehl aus, um eine Revision zu erstellen, die einen anderen Schlüssel verwendet.

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/new-key-id \
    --build-config '{
      "codeConfig": { "s3Object": { "bucket": "my-bucket", "key": "my-app-
v2.zip" } },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role"
    }'
```

2. Aktualisieren Sie die Revisionsgewichte des Endpunkts, um den Datenverkehr an die neue Version weiterzuleiten. Weitere Informationen finden Sie unter the section called “Bereitstellungsmuster”.

Einschränkung des Zugriffs auf Ihren KMS-Schlüssel

Die Schlüsselrichtlinie in the section called “Beispiel für eine Schlüsselrichtlinie” schränkt den Zugriff mit den folgenden Bedingungsschlüsseln ein. Jede dieser Optionen schränkt die Verwendung des Schlüssels ein, sodass der Zugriff auf den Schlüssel nicht umfassender ist, als für die Webfunktion erforderlich ist.

Verschlüsselungskontext

Lambda legt den Verschlüsselungskontext `aws:lambda:WebFunctionRevisionArn` bei jeder AWS KMS Revisionsanforderung fest. Verwenden Sie eine `StringLike` Bedingung

auf`kms:EncryptionContext:aws:lambda:WebFunctionRevisionArn`, um den Schlüssel auf Revisionen einer bestimmten Webfunktion zu beschränken. Wenden Sie diese Bedingung sowohl auf die Creator-Anweisung als auch auf die Lambda-Serviceprinzipal-Anweisung an.

Wenden Sie diese Bedingung nicht auf eine Erklärung an, die gewährt`kms:DescribeKey`. Für diesen Vorgang gibt es keinen Verschlüsselungskontext, daher würde die Bedingung ihn ablehnen.

Das folgende Beispiel zeigt die Creator-Anweisung mit der Bedingung für den Verschlüsselungskontext.

```
{
    "Sid": "Allow the revision creator to generate and validate data keys",
    "Effect": "Allow",
    "Principal": { "AWS": "arn:aws:iam::111122223333:role/MyWebFunctionDeployRole" },
    "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "kms:EncryptionContext:aws:lambda:WebFunctionRevisionArn":
 "arn:aws:lambda:us-east-1:111122223333:web-function/my-function/revision/*"
        }
    }
}
```

kms: ViaService Zustand

Die `kms:ViaService` Bedingung schränkt den Schlüssel ein, sodass ihn nur Anfragen verwenden können, die Lambda in Ihrem Namen stellt. Der Wert ist Region-specific:. `lambda.us-east-1.amazonaws.com`

Mit dieser Bedingung werden direkte Aufrufe AWS KMS verweigert, die mit denselben Anmeldeinformationen getätigt wurden (z. B. wenn Sie es `aws kms generate-data-key` selbst ausführen), aber das Erstellen einer Webfunktion ist erfolgreich, da Lambda die AWS KMS Aufrufe für Sie durchführt.

AWS KMS verwendet keine gemeinsamen Richtlinien für verknüpfte Schlüssel aus mehreren Regionen. Jeder Replikatschlüssel hat seine eigene Richtlinie. Wenn Sie die `kms:ViaService` Bedingung mit einem Schlüssel für mehrere Regionen verwenden, verwenden Sie den regionalen Dienstnamen, der der Region jedes Schlüssels entspricht (z. B.

`lambda.us-east-1.amazonaws.com` in der Schlüsselrichtlinie us-east-1 und `lambda.us-west-2.amazonaws.com` in der replikatschlüsselrichtlinie us-west-2). Sie können auch mehrere regionale Werte in einer Bedingung auflisten, da `StringEquals` eine Reihe von Werten akzeptiert wird.

Das folgende Beispiel zeigt die Creator-Anweisung sowohl mit der Bedingung als auch mit der `kms:ViaService` Bedingung für den Verschlüsselungskontext.

```
{
    "Sid": "Allow the revision creator to generate and validate data keys",
    "Effect": "Allow",
    "Principal": { "AWS": "arn:aws:iam::111122223333:role/MyWebFunctionDeployRole" },
    "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "kms:EncryptionContext:aws:lambda:WebFunctionRevisionArn":
 "arn:aws:lambda:us-east-1:111122223333:web-function/my-function/revision/*"
        },
        "StringEquals": {
            "kms:ViaService": "lambda.us-east-1.amazonaws.com"
        }
    }
}
```

Verwirrter Abgeordneter, Schutz

Das Problem des verwirrten Stellvertreters tritt auf, wenn eine Entität, die nicht berechtigt ist, eine Aktion auszuführen, eine Entität mit mehr Rechten dazu zwingt. Da der Lambda Service Principal Ihren vom Kunden verwalteten Schlüssel verwenden kann, beschränken Sie diesen Zugriff auf die Web Function-Revisionen, die Sie besitzen. Weitere Informationen finden Sie unter Das Problem des verwirrten Stellvertreters im IAM-Benutzerhandbuch.

Wenn Lambda AWS KMS mit seinen eigenen Dienstanmeldedaten aufruft, um eine Ausführungsumgebung zu starten, sendet es den Revision-ARN als Quell-ARN und Ihre AWS-Konto ID als Quellkonto. Verwenden Sie den Schlüssel aws: SourceArn condition in der Lambda Service Principal-Anweisung, um den Schlüssel auf Revisionen einer bestimmten Webfunktion zu beschränken.

```
{
    "Sid": "Allow Lambda to decrypt the data key for Web Function revisions",
    "Effect": "Allow",
    "Principal": { "Service": "lambda.amazonaws.com" },
    "Action": "kms:Decrypt",
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "aws:SourceArn": "arn:aws:lambda:us-east-1:111122223333:web-function/my-
function/revision/*"
        }
    }
}
```

Um den Schlüssel auf Ihr Konto und nicht auf eine bestimmte Webfunktion zu beschränken, verwenden Sie den SourceAccount Bedingungsschlüssel aws: mit Ihrer AWS-Konto ID.

Wenden Sie diese Bedingung auf die Lambda-Serviceprinzipalanweisung an. Die `kms:ViaService` unter beschriebene Bedingung the section called “kms: ViaService Zustand” schränkt ein, welcher Dienst den Schlüssel verwenden kann, bietet aber keinen Schutz für verwirrte Stellvertreter.

Verhalten bei Fehlern

Lambda validiert den AWS KMS Schlüssel, wenn Sie eine Revision erstellen. Die folgenden Fehler können bei der Erstellung auftreten:

- Validierungsfehler — Der Schlüssel-ARN ist ungültig, der Schlüssel existiert nicht, der Schlüssel ist deaktiviert oder der Schlüssel ist kein symmetrischer Verschlüsselungsschlüssel.
- Fehler „Zugriff verweigert“ — Der aufrufende Principal oder der Lambda-Serviceprinzipal verfügt nicht über die erforderlichen AWS KMS Berechtigungen für den Schlüssel.

⚠ Deaktivieren oder entziehen Sie nicht den Zugriff auf den Schlüssel einer aktiven Revision

Wenn Sie den AWS KMS Schlüssel deaktivieren oder dem Lambda-Serviceprinzipal nach einer Revision den Zugriff entziehen`Active`, kann Lambda den Datenschlüssel nicht mehr entschlüsseln. Dadurch wird verhindert, dass Lambda neue Ausführungsumgebungen für diese Revision startet, wodurch Aufrufe unterbrochen werden. Löschen Sie außerdem keine Replikatschlüssel in Regionen, in denen die Revision noch bereitgestellt wird. Dies hat den gleichen Effekt (Lambda kann in dieser Region keine neuen Ausführungsumgebungen

starten). Um den Dienst wiederherzustellen, erstellen Sie eine neue Version mit einem zugänglichen Schlüssel oder aktivieren Sie den ursprünglichen Schlüssel erneut und stellen Sie die Schlüsselrichtlinie wieder her.

Verschlüsselung mit MultiRegion Endpunkten PerRegion

Bei Webfunktionen mit MultiRegion oder PerRegion Endpunkten überprüft Lambda während der Replikation die Schlüsselverfügbarkeit in jeder Region. Wenn das Schlüsselreplikat für mehrere Regionen in einer Zielregion fehlt oder nicht darauf zugegriffen werden kann, schlägt die Replikation nur für diese Region fehl. Andere Regionen funktionieren weiterhin normal.

Sie können den Bundesstaat jeder Region einsehen und den Status aktualisieren, indem Sie anrufen`GetWebFunctionEndpoint`. Der Eintrag jeder Region `regionalEndpoints` enthält `state` und `stateReason` (den Bereitstellungsstatus der Region) `updateStatus` und und `updateStatusReason` (das Ergebnis der letzten Aktualisierung). Das Verhalten unterscheidet sich je nachdem, ob der Endpunkt erstellt oder aktualisiert wird:

- Während der Endpunkterstellung — Der Eintrag in der Region `regionalEndpoints` zeigt ein `state` Von `Failed` und ein`stateReason`, das das Hauptproblem beschreibt. Der Gesamtendpunkt wird `state` ebenfalls angezeigt `Failed` und `stateReason` leitet Sie zu den regionalen Einträgen weiter.
- Während eines Endpunkt-Updates — Die Region wird `Failed` mit einer Anzeige `updateStatus` angezeigt`updateStatusReason`, die das Hauptproblem beschreibt. Der allgemeine Endpunkt wird `updateStatus` ebenfalls angezeigt `Failed` und `updateStatusReason` leitet Sie zu den regionalen Einträgen weiter.

Um eine fehlerhafte Region zu beheben, erstellen Sie ein Schlüsselreplikat in dieser Region oder stellen Sie den Zugriff auf das vorhandene Replikat wieder her und aktualisieren Sie dann den Endpunkt, um die Replikation erneut zu versuchen.

Überwachung der Interaktion der Webfunktionen mit AWS KMS

Sie können CloudTrail damit die AWS KMS Anfragen verfolgen, die Lambda in Ihrem Namen stellt, wenn Sie einen vom Kunden verwalteten Schlüssel mit einer Webfunktion verwenden. Informationen zum Auffinden von AWS KMS Ereignissen in Ihren CloudTrail Protokollen finden Sie unter Suchen von CloudTrail Protokollen nach AWS KMS API-Aufrufen im AWS Key Management Service Entwicklerhandbuch.

In der folgenden Tabelle sind die AWS KMS Ereignisse aufgeführt, die von Webfunktionen generiert werden, sowie die Werte, die Sie bei jedem Ereignis überprüfen können.

eventName	Wann es auftritt	Zu prüfende Werte
`DescribeKey`	Wenn Sie eine Webfunktion oder Revision mit einem vom Kunden verwalteten Schlüssel erstellen.	`userIdentity` ist der IAM-Prinzipal, der die Revision erstellt hat. Dieses Ereignis hat keinen Verschlüsselungskontext, da `DescribeKey` es sich um einen Metadatenvorgang handelt.
`GenerateDataKey`	Wenn Sie eine Webfunktion oder Revision erstellen. Lambda generiert den Datenschlüssel, der das Codeartefakt und die Umgebungsvariablen der Revision verschlüsselt.	`userIdentity` ist der IAM-Prinzipal, der die Revision erstellt hat. `encryptionContext` enthält `aws:lambda:WebFunctionRevisionArn` .
`Decrypt`	Wenn Sie eine Webfunktion oder Revision erstellen, validiert Lambda den Schlüsselzugriff. Lambda ruft diese Operation auch auf, wenn es die Revision in einer Replikatregion bereitstellt, wenn es eine neue Ausführungsumgebung startet und wenn ein Aufrufer die Umgebungsvariablen der Revision mit liest. `GetWebFunctionRevision`	Bei Aufrufen zur Erstellungszeit `userIdentity` ist dies der IAM-Prinzipal, der die Revision erstellt hat. Lambda führt auch einen Probelauf durch, `Decrypt` `lambda.amazonaws.com` um zu bestätigen, dass der Dienstprinzipal den Schlüssel verwenden kann. Für Runtime-Aufrufe `userIdentity` ist der Lambda-Service Principal (`lambda.amazonaws.com`). Denn `GetWebFunctionRevision` Lambda ruft `Decrypt` mit den Anmeldeinformationen des Anrufers an. `encryptionContext` enthält. `aws:lambda:WebFunctionRevisionArn`

Die `Decrypt` Ereignisse `GenerateDataKey` und beinhalten den Verschlüsselungskontext `aws:lambda:WebFunctionRevisionArn` mit dem Revision-ARN. Verwenden Sie diesen Wert, um AWS KMS Anfragen für eine bestimmte Webfunktions-Revision zu identifizieren.

Überwachung von Lambda-Webfunktionen

AWS Lambda Web Functions veröffentlichen Metriken und Protokolle auf Amazon CloudWatch. Verwenden Sie sie, um Leistungsengpässe zu identifizieren, Fehler zu debuggen und Alarme einzurichten, die Sie benachrichtigen, wenn die Fehlerraten bei Bereitstellungen auf Canary oder bei Verkehrsverschiebungen steigen.

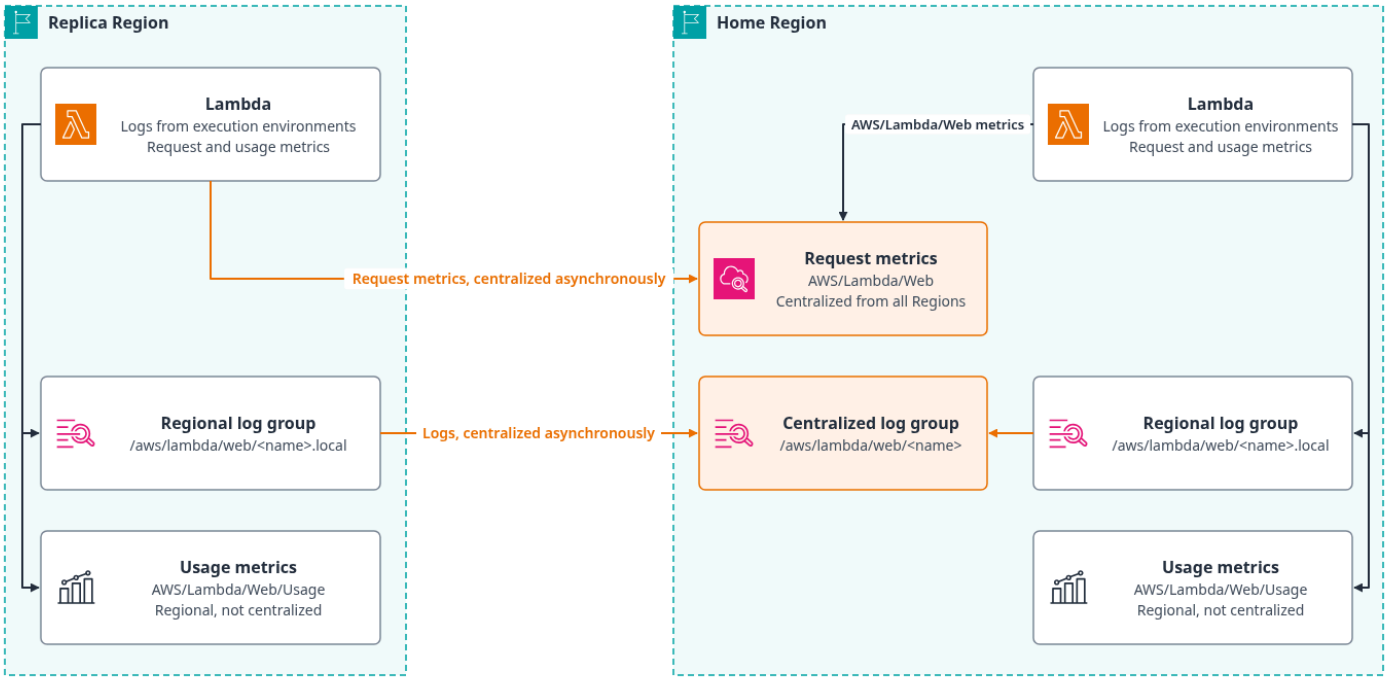
Für eine Webfunktion, die Anfragen in mehr als einer bearbeitet AWS-Region, sammelt Lambda Metriken und Protokolle in jeder Region, in der Anfragen ausgeführt werden. Es zentralisiert die Anforderungsmetriken und protokolliert sie in der Heimatregion der Webfunktion, sodass Sie die gesamte Anwendung von einem Ort aus überwachen können. Nutzungs- und Drosselungsmetriken bleiben in der Region, in der sie ausgegeben werden, da die darin beschriebenen Kontingente für jede Region gelten.

Jede Art von Telemetrie beantwortet eine andere Frage:

- Metriken — Verfolgen Sie, wie Ihre Webfunktion im Laufe der Zeit abschneidet, und melden Sie bei Problemen Alarm. Details hierzu finden Sie unter the section called “Metriken”.
- Protokolle — Untersuchen Sie einzelne Anfragen und Fehler. Details hierzu finden Sie unter the section called “Protokolle”.
- Benutzerdefinierte Metriken — Veröffentlichen Sie Ihre eigenen Anwendungsmetriken, indem Sie das eingebettete Metrikformat in Ihre Logs schreiben. Details hierzu finden Sie unter the section called “Benutzerdefinierte Metriken”.
- CloudTrail— Zeichnen Sie API-Aufrufe für Ihre Webfunktionen auf, einschließlich Aufrufe, die als Datenereignisse CloudTrail protokolliert werden. Details hierzu finden Sie unter the section called “CloudTrail protokolliert”.

Verwenden Sie zur Fehlerbehebung in der Region, in der eine Anfrage ausgeführt wurde, die Protokollgruppe Regional`/aws/lambda/web/`*`function-name`*`.local`, in der Ausführungsumgebungen Protokolle schreiben. Lambda zentralisiert die Protokolle asynchron in der `/aws/lambda/web/`*`function-name`* Protokollgruppe in der Heimatregion. Hinweise zu Heimatregionen und Replikation finden Sie unter. the section called “Multi-Region Einsatz”

Das folgende Diagramm zeigt, wie Lambda Telemetrie über Regionen hinweg schreibt und zentralisiert. Anforderungsmetriken im `AWS/Lambda/Web` Namespace und die Protokolle werden in der Heimatregion zentralisiert. Die Nutzungsmetriken im `AWS/Lambda/Web/Usage` Namespace verbleiben in jeder Region.



Themen

- Abrechnung per Telemetrie
- Überlegungen zur Telemetrie
- Verwenden von CloudWatch Metriken mit Lambda-Webfunktionen
- Webfunktionsprotokolle in Amazon CloudWatch Logs
- Veröffentlichen von benutzerdefinierten Metriken mit dem eingebetteten Metrikformat
- Protokollieren von API-Aufrufen für Webfunktionen mit AWS CloudTrail

Abrechnung per Telemetrie

Telemetrie für Webfunktionen wird wie folgt abgerechnet:

Telemetrie	Fakturierung
Von Lambda veröffentlichte Metriken	Kostenlos. Lambda veröffentlicht die Metriken in den `AWS/Lambda/Web/Usage` Namespaces `AWS/Lambda/Web` und ist für Sie kostenlos.
Protokolle	Sie zahlen die Standardgebühren von Amazon CloudWatch Logs für das Schreiben und Speichern von Protokollen in der `.local` Protokollgruppe in jeder Region. Sie zahlen auch für das Speichern der Kopien in der zentralen Protokollgruppe in der Heimatregion. Lambda kopiert jedes Protokollereignis kostenlos in die Heimatregion. Die erste Zentralisierung des Protokollereignisses ist kostenlos.
Benutzerdefinierte Metriken	Für benutzerdefinierte Metriken, die Sie mit dem eingebetteten Metrikformat veröffentlichen, fallen Amazon CloudWatch Logs-Aufnahmegebühren und Gebühren für CloudWatch benutzerd efinierte Kennzahlen an. Die erste Zentralisierung des Protokoll ereignisses ist kostenlos. Weitere Informationen finden Sie unter the section called “Benutzerdefinierte Metriken”.

Aktuelle Preise finden Sie unter CloudWatchAmazon-Preise.

Überlegungen zur Telemetrie

Informieren Sie sich über die folgenden Verhaltensweisen, bevor Sie Dashboards, Alarme oder Automatisierungen für die Telemetrie von Web Function erstellen.

AWS X-Ray Nachverfolgung

Wenn eine Anfrage einen `X-Amzn-Trace-Id` Header enthält, leitet Lambda ihn unverändert an Ihre Anwendung weiter. Lambda fügt Anfragen, die keinen enthalten, keinen Trace-Header hinzu, und Web Functions erstellen keine Lambda-Servicesegmente. Da Anrufer diesen Header setzen, behandeln Sie seinen Wert als Clienteingabe. Um Traces aufzuzeichnen, senden Sie Trace-Daten aus Ihrem Code direkt an X-Ray, z. B. mit der AWS Distribution für OpenTelemetry, und geben Sie Ihrer Ausführungsrolle Schreibzugriff. X-Ray

Latenz bei der Zentralisierung

Metriken und Protokolle erreichen die Heimatregion asynchron. Im Steady-State-Modus zentralisiert Lambda Metriken und Protokolle in der Regel innerhalb weniger Minuten. Telemetrie kann später während einer Betriebsunterbrechung eintreffen.

Wenn Sie einen Endpunkt erstellen, der in mehr als einer Region ausgeführt wird, richtet Lambda die Metrikzentralisierung für jede Replikatregion ein. CloudWatch Es kann mehrere Minuten dauern, bis der Endpunkt in einer Region aktiv wird, bis mit der Zentralisierung der Metriken aus dieser Region begonnen wird. Die zuvor ausgegebenen Metriken verbleiben dann nur in dieser Region und werden später nicht zur Heimatregion hinzugefügt.

Berücksichtigen Sie diese Latenz bei der Bereitstellungsautomatisierung. Wenn eine Pipeline eine Webfunktion oder einen Endpunkt erstellt und dann sofort Metriken oder Protokolle in der Heimatregion überprüft, sollten Sie Zeit bis zum Eintreffen der ersten Telemetriedaten einplanen, anstatt deren Fehlen als Fehler zu behandeln.

Telemetrie in jeder Region

Lambda schreibt Telemetrie in jede Region, in der Anfragen ausgeführt werden, und zentralisiert dann einen Teil davon in der Heimatregion. Worauf Sie sich in jeder Region verlassen können, hängt von der Art der Telemetrie ab:

Metriken anfordern () `AWS/Lambda/Web`

> Lambda kann diese Metriken auch in anderen Regionen schreiben, in denen Anfragen ausgeführt werden. Sie können ohne vorherige Ankündigung geändert oder entfernt werden. Wir empfehlen Ihnen daher, sich nicht auf sie zu verlassen und stattdessen die zentralen Werte in der Heimatregion für Alarme, Dashboards und Automatisierung zu verwenden. Die zentralisierten Werte beinhalten Anfragen aus allen Regionen, die der Endpunkt bedient.

Nutzungs- und Drosselungsmetriken () `AWS/Lambda/Web/Usage`

> Diese Metriken existieren nur in der Region, in der die Anfrage bearbeitet wurde. Sie sind in jeder Region alarmiert. Details hierzu finden Sie unter the section called “Regionale Nutzungs- und Drosselungsmetriken”.

Protokolle

> In jeder Region enthält die `.local` Protokollgruppe nur die Protokolle aus dieser Region. Diese Protokolle sind vor Abschluss der Zentralisierung verfügbar und werden nicht durch Verzögerungen bei der Zentralisierung beeinträchtigt, was sie für die Behebung regionaler

Probleme nützlich macht. Die zentralisierte Protokollgruppe in der Heimatregion enthält Protokolle aus allen Regionen. Erstellen Sie gemeinsame Abfragen, Abonnements und Aufbewahrungsrichtlinien für die zentralisierte Protokollgruppe. Details hierzu finden Sie unter the section called “Protokollgruppen”.

Betriebsunterbrechungen

Störungen der Infrastruktur, wie z. B. ein Ausfall der Availability Zone, können die Telemetrie verzögern, und einige Messwerte und Protokollereignisse können verloren gehen. Wenn die Heimatregion beeinträchtigt ist, können zentralisierte Messwerte und Protokolle in der Heimatregion verzögert oder nicht verfügbar sein. Nutzungsmetriken und `.local` Protokollgruppen in anderen Regionen hängen nicht von der Zentralisierung ab und empfangen weiterhin Telemetriedaten aus diesen Regionen.

Telemetrie wird nach bestem Wissen und Gewissen ausgegeben. In seltenen Fällen kann die Telemetrie gelöscht oder dupliziert werden. Behandeln Sie die Web-Funktionstelemetrie nicht als Prüfprotokoll. Um Konfigurationsänderungen und Aufrufe zu überprüfen, verwenden Sie. AWS CloudTrail Weitere Informationen finden Sie unter the section called “CloudTrail protokolliert”.

Erforderliche -Berechtigungen

Für die Protokollierung sind die Amazon CloudWatch Logs-Berechtigungen `logs:CreateLogGroup` und `logs:PutLogEvents` die Ausführungsrolle Ihrer Web-Funktion erforderlich. `logs:CreateLogStream` Ohne sie kann Lambda keine Protokollgruppen erstellen oder Protokollereignisse schreiben, und die Ausgabe Ihrer Anwendung geht verloren. Eine Beispielrichtlinie finden Sie unter the section called “Rolle „Ausführung“”.

Lambda veröffentlicht die Metriken in den `AWS/Lambda/Web/Usage` Namespaces `AWS/Lambda/Web` und selbst, sodass sie keine Genehmigung in der Ausführungsrolle benötigen, einschließlich. `cloudwatch:PutMetricData` Das eingebettete Metrikformat wird ebenfalls nicht `PutMetricData` direkt aufgerufen, erfordert jedoch dieselben Amazon CloudWatch Logs-Berechtigungen, da Lambda die Metriken aus Ihren Protokollereignissen extrahiert.

Verwenden von CloudWatch Metriken mit Lambda-Webfunktionen

AWS Lambda veröffentlicht Web Function-Metriken CloudWatch in zwei Namespaces auf Amazon:

- `AWS/Lambda/Web`— Anforderungs- und Antwortmetriken für jeden Endpunkt sowie Kennzahlen zur Ressourcennutzung für die Ausführungsumgebungen, die jede Webfunktion bedienen.

Lambda zentralisiert diese Metriken in der Heimatregion der Webfunktion, sodass die Werte in der Heimatregion den Verkehr aus allen Regionen enthalten, die der Endpunkt bedient.

- `AWS/Lambda/Web/Usage`— Metriken zur regionalen Nutzung und Drosselung, die den Verbrauch anhand regionaler Kontingente und Endpunktgrenzwerte aufzeigen. Diese Kennzahlen sind nicht zentralisiert. Sie verbleiben in der Region, in der die Anfrage zugestellt wurde.

Lambda veröffentlicht Anforderungs-, Nutzungs- und Drosselungsmetriken nur, wenn eine Anfrage einen Wert für sie generiert. Die Metriken der Ausführungsumgebung sind unterschiedlich: Lambda veröffentlicht sie jede Minute für jede aktive Ausführungsumgebung. Einige Metriken sind spärlich: Lambda veröffentlicht keinen Datenpunkt von 0 für einen Zeitraum, in dem das Ereignis nicht eingetreten ist. Wenn beispielsweise keine Anfrage in einem Zeitraum eine 5xx-Antwort zurückgibt, `HTTPResponseStatusClass5XX` gibt es keinen Datenpunkt für diesen Zeitraum. In den folgenden Tabellen wird beschrieben, wann Lambda die einzelnen Metriken ausgibt. Wenn Sie Alarme für spärliche Metriken erstellen, konfigurieren Sie, wie der Alarm mit fehlenden Daten umgeht. Weitere Informationen finden Sie unter Konfiguration der Behandlung fehlender Daten durch CloudWatch Alarme im CloudWatch Amazon-Benutzerhandbuch.

Themen

- Anforderungsmetriken
- Metriken der Ausführungsumgebung
- Regionale Nutzungs- und Drosselungsmetriken
- Metriken mit anderen CloudWatch Funktionen verwenden
- Einen Alarm bei 5xx Antworten einrichten

Anforderungsmetriken

Anforderungsmetriken werden im `AWS/Lambda/Web` Namespace mit den `EndpointName` Dimensionen `FunctionName` und veröffentlicht. Sie verwenden das `Count` Gerät. Benutze die `Sum` Statistik.

Metrik	Statistik	Wenn Lambda es aussendet	Description
`ExecutedRequests`	`Sum`	Nur für Zeiträume, in denen mindestens eine	Die Anzahl der Anfragen, die eine Ausführungsumgebung erreicht

Metrik	Statistik	Wenn Lambda es aussendet	Description
		Anfrage eine Ausführun gsumgebung erreicht hat.	haben. Anfragen, die Lambda drosselt oder ablehnt, bevor sie eine Ausführungsumgebung erreichen, werden nicht gezählt.
`HTTPRespo nseStatus Class4XX`	`Sum`	Spärlich. Nur für Zeiträume mit mindestens einer 4xx-Antwort.	Die Anzahl der Antworten, die an Anrufer mit einem Statuscode zwischen 400 und 499 zurückges endet wurden. Die Antwort kann von Ihrer Anwendung oder von Lambda kommen, einschließlich gedrosselter Anfragen, die HTTP 429 empfangen. Anfragen, die Lambda ablehnt, bevor es den Endpunkt identifiziert, wie z. B. falsch formatierte Anfragen, werden nicht gezählt.
`HTTPRespo nseStatus Class5XX`	`Sum`	Spärlich. Nur für Zeiträume mit mindestens einer 5xx-Antwort.	Die Anzahl der Antworten, die an Anrufer mit einem Statuscode zwischen 500 und 599 zurückges endet wurden. Die Antwort kann von Ihrer Anwendung oder von Lambda kommen. Während einer Serviceun terbrechung kann diese Kennzahl Fehler unterschätzen.

Bei den Statusklassen-Metriken wird der Statuscode gezählt, den der Anrufer erhalten hat. Sie unterscheiden nicht zwischen Antworten, die von Ihrer Anwendung geschrieben wurden, und Antworten, die von Lambda zurückgegeben wurden. Wenn Ihre Anwendung den Antwortstatus und die Header bereits gesendet hat, wird ein späterer Fehler, z. B. ein unterbrochener Stream oder ein Anforderungs-Timeout, unter dem ursprünglichen Statuscode gezählt. Verwenden Sie den `lambda-web-error-type` Antwort-Header und Ihre Logs, um die Ursache eines Fehlers zu ermitteln. Um Anwendungsfehler getrennt zu zählen, veröffentlichen Sie Ihre eigenen Messwerte. Weitere Informationen finden Sie unter the section called “Benutzerdefinierte Metriken”.

Um die Serverfehlerrate zu schätzen, dividieren Sie mit `ExecutedRequests` Hilfe CloudWatch metrischer Berechnungen `HTTPResponseStatusClass5XX` durch die Werte für denselben Zeitraum. Verwenden Sie beispielsweise`FILL(m5xx, 0) / m_executed`, where `m5xx` und `m_executed` are die IDs der beiden Messwerte. Da `HTTPResponseStatusClass5XX` es dünn gesät ist, verwenden Sie `FILL` es so, dass Perioden ohne Fehler 0 ergeben, anstatt keinen Wert zu haben. `HTTPResponseStatusClass5XX`zählt auch 5xx-Antworten, die Lambda zurückgegeben hat, bevor eine Anfrage eine Ausführungsumgebung erreichte. `ExecutedRequests`zählt diese Antworten nicht, sodass das Verhältnis die Rate während einer Betriebsunterbrechung überbewerten kann.

Anforderungsmetriken in der Heimatregion aggregieren den Verkehr aus allen Regionen, die der Endpunkt bedient, sodass ein Alarm in der Heimatregion den gesamten Endpunkt abdeckt. Erstellen Sie Alarme und Dashboards für die Werte der Heimatregion. Verlassen Sie sich in anderen Regionen nicht auf diese Kennzahlen. Weitere Informationen finden Sie unterthe section called “Telemetrie in jeder Region”.

Lambda gibt Anforderungsmetriken aus, wenn eine Anfrage abgeschlossen ist. Bei einer Anfrage mit langer Laufzeit kann der Datenpunkt bis zum Timeout der Anfrage nach dem Start der Anfrage angezeigt werden, sodass Fehler aufgrund langer Anfragen Ihre Alarme erst spät erreichen können. Informationen zum Zeitlimit für Anfragen finden Sie unter. the section called “Zeitüberschreitung”

Metriken der Ausführungsumgebung

Metriken zur Ausführungsumgebung beschreiben die Ressourcen, die die Ausführungsumgebungen verwenden, die Ihre Webfunktion bedienen. Lambda veröffentlicht sie im `AWS/Lambda/Web` Namespace mit der `FunctionName` Dimension und zentralisiert sie in der Heimatregion der Webfunktion.

Lambda gibt diese Metriken einmal pro Minute für jede aktive Ausführungsumgebung aus, sodass sie spärlich sein könnten, wenn es keine aktiven Ausführungsumgebungen gibt. Ein Wert kann 0 sein. Eine Ausführungsumgebung kann mehr als einen Datenpunkt pro Minute veröffentlichen, z. B. wenn sie initialisiert und heruntergefahren wird. Verwenden Sie die `SampleCount` Statistik also nicht, um aktive Umgebungen zu zählen. Um sie für einen Endpunkt zu zählen, verwenden Sie. `EndpointActiveEnvironments`

Metrik	Einheit	Statistik	Description
`Environme ntActiveTime`	`Millisec nds`	`Sum`	Die Uhrzeit, zu der die Ausführungsumgebun gen der Funktion aktiv waren, einschließlich

Metrik	Einheit	Statistik	Description
			der Initialisierung. Jeder Datenpunkt entsprich t der aktiven Zeit einer Umgebung seit ihrem vorherigen Datenpunkt. Die Summe über einen Zeitraum entspricht also der gesamten verbrauchten Umgebungszeit.
`Environme ntCPUTime`	`Millisec nds`	`Sum`	Die vCPU-Zeit, die die Ausführungsumgebun gen der Funktion verbraucht haben, einschlie ßlich der Initialisierung.
`Environme ntMaxMemo ryUsage`	`Megabyte`	`Maximum`	Der maximale Speicherverbrauch, den eine Ausführungsumgebung in ihrer Lebensdauer bisher genutzt hat.
`Environme ntConcurr entRequests`	`Count`	`Maximum`	Die höchste Anzahl von Anfragen, die eine Ausführungsumgebung während des Zeitraums gleichzeitig bearbeitet hat.
`ResponseBytes`	`Bytes`	`Sum`	Die Anzahl der Antwortbytes, die die Ausführun gsumgebungen der Funktion gesendet haben.

Die Metriken der Ausführungsumgebung enthalten die `EndpointName` Dimension nicht. Sie decken alle Endpunkte der Webfunktion ab.

Regionale Nutzungs- und Drosselungsmetriken

Nutzungs- und Drosselungsstatistiken zeigen, wie nah Ihr Konto und Ihre Endgeräte an ihren regionalen Grenzwerten sind und welches Limit eine Anfrage gedrosselt hat. Das Kontingent für die Kontoanforderungsrate und das Arm64-vCPU-Kontingent für das Konto gelten jeweils separat AWS-Region, sodass Lambda diese Metriken in jeder Region veröffentlicht und nicht zentralisiert. Um die Nutzung für eine Region zu sehen, sehen Sie sich die Metriken in dieser Region an. Die Kontingente finden Sie unterthe section called “Lambda-Web-Funktionen”.

Diese Metriken werden im `AWS/Lambda/Web/Usage` Namespace mit den `ExecutionRegion` Dimensionen `FunctionNameEndpointName`, und veröffentlicht. Sie verwenden das `Count` Gerät.

Metrik	Statistik	Wenn Lambda es aussendet	Description
`ExecutedRequests`	`Sum`	Nur für Zeiträume, in denen mindestens eine Anfrage eine Ausführungsumgebung in der Region erreicht hat.	Die Anzahl der Anfragen, die eine Ausführungsumgebung in der Region erreicht haben.
`EndpointActiveEnvironments`	`Maximu`	Mit Anfragen an den Endpunkt. Der Wert kann 0 sein. Es gibt keine Datenpunkte, und der Endpunkt empfängt keine Anfragen.	Die Anzahl der aktiven Ausführungsumgebungen des Endpunkts in der Region. Vergleichen Sie diesen Wert mit`maxEnvironments` .
`AccountRequestThrottles`	`Sum`	Spärlich. Nur für Zeiträume mit mindestens einer solchen Drosselung.	Die Anzahl der Anfragen an den Endpunkt, die gedrosselt wurden, weil das Konto seine Quote für die Anforderungsrate in der Region erreicht hat.
`EndpointRequestThrottles`	`Sum`	Spärlich. Nur für Zeiträume mit mindestens einer solchen Drosselung.	Die Anzahl der Anfragen, die gedrosselt wurden, weil der Endpunkt sein Limit für die Anforderungsrate () erreicht hat. `rateLimit`
`AccountActiveVCPUArm64Throttles`	`Sum`	Spärlich. Nur für Zeiträume mit mindestens einer solchen Drosselung.	Die Anzahl der Anfragen an den Endpunkt, die gedrosselt wurden, weil ihre Bearbeitung das Arm64-vCPU-Kontingent des Kontos in der Region überschritten hätte. Wie bei den anderen Drosselungsmetriken werden nur die Drosselungen dieses Endpunkts gezählt. Um

Metrik	Statistik	Wenn Lambda es aussendet	Description
			die Gesamtzahl der Konten zu sehen, summieren Sie sie über die Endpunkte hinweg.
`EndpointActiveEnvironmentThrottles`	`Sum`	Spärlich. Nur für Zeiträume mit mindestens einer solchen Drosselung.	Die Anzahl der Anfragen, die gedrosselt wurden, weil ihre Bearbeitung die Endpunkte instellung überschritten hätte. `maxEnvironments`

Lambda ordnet jeder gedrosselten Anfrage ein Limit zu, sodass eine gedrosselte Anfrage höchstens eine Drosselungsmetrik erhöht. Anfragen, die Lambda drosselt, weil zu viele Ausführungsumgebungen gleichzeitig initialisiert werden, geben HTTP 429 mit dem `Platform.EndpointConcurrentInitThrottle` Fehlertyp zurück, erhöhen jedoch keine Drosselungsmetrik. Gedrosselte Anfragen werden auch im Namespace berücksichtigt. `HTTPResponseStatusClass4XX` `AWS/Lambda/Web`

Um einen `PerRegion` Or-Endpunkt zu überwachen, können Sie sich die Nutzungsmetriken in jeder Region anzeigen lassen `MultiRegion` oder einen Alarm ausgeben oder sie in einem CloudWatch regionsübergreifenden Dashboard kombinieren.

Metriken mit anderen CloudWatch Funktionen verwenden

Da `AWS/Lambda/Web` Metriken in der Heimatregion zentralisiert sind, können Sie sie mit den CloudWatch Standardfunktionen in dieser Region verwenden, darunter:

- Amazon CloudWatch Observability Access Manager — Teilen Sie Web Function-Metriken mit einem zentralen Überwachungskonto, sodass ein Konto viele Anwendungskonten beobachten kann.
- Dashboards und Alarme — Erstellen Sie Dashboards und Alarme zu den Kennzahlen der Heimatregion, um den gesamten Einsatz in mehreren Regionen abzudecken.

Einen Alarm bei 5xx Antworten einrichten

Erstellen Sie CloudWatch Alarme, um Probleme wie erhöhte Fehlerraten oder Drosselung zu erkennen.

Mit dem folgenden Befehl wird ein Alarm ausgelöst, der ausgelöst wird, wenn Anrufer innerhalb von fünf Minuten mehr als fünf 5xx-Antworten vom Endpunkt erhalten.

```
aws cloudwatch put-metric-alarm \
    --alarm-name my-web-app-prod-5xx \
    --namespace AWS/Lambda/Web \
    --metric-name HTTPResponseStatusClass5XX \
    --statistic Sum \
    --period 300 \
    --threshold 5 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 1 \
    --treat-missing-data notBreaching \
    --dimensions Name=FunctionName,Value=my-web-app Name=EndpointName,Value=prod \
    --alarm-actions arn:aws:sns:us-east-1:111122223333:my-alerts
```

Ersetzen Sie die folgenden Werte:

- *my-web-app-prod-5xx*— Ein Name für den Alarm.
- *my-web-app*— Der Name Ihrer Webfunktion.
- *prod* – Der Name des Endpunkts.
- *arn:aws:sns:us-east-1:111122223333:my-alerts*— Der ARN des Amazon SNS SNS-Themas, das Alarmbenachrichtigungen empfängt.

`--treat-missing-data notBreaching`ist hier wichtig. `HTTPResponseStatusClass5XX`ist eine spärliche Metrik, sodass Perioden ohne 5xx-Antworten keine Datenpunkte ergeben. Ohne diese Einstellung würde der Alarm während dieser Zeiträume in den `INSUFFICIENT_DATA` Status wechseln.

> **Note**
>
> Metriken für Web-Funktionen enthalten keine Revisionsdimension. Um einer Revision während einer Canary-Implementierung Fehler zuzuordnen, legen Sie eine Umgebungsvariable fest, die den Build identifiziert, z. B. `APP_VERSION` in jeder Revision,

und nehmen Sie seinen Wert in Ihre strukturierten Logs oder als Dimension in Metriken auf, die Sie mit dem eingebetteten Metrikformat veröffentlichen. Systemprotokollereignisse zeichnen beispielsweise `platform.initStart` auch auf, wie `revisionId` die einzelnen Ausführungsumgebungen ausgeführt werden, und der `lambda-web-logstream` Antwort-Header identifiziert den Protokollstream der Ausführungsumgebung, die eine Anfrage bearbeitet hat. Informationen zum Rollback-Verfahren finden Sie unterthe section called “Wird zurückgesetzt”.

Webfunktionsprotokolle in Amazon CloudWatch Logs

AWS Lambda erfasst alles, was Ihre Anwendung in die Standardausgabe und den Standardfehler schreibt, und sendet es zusammen mit CloudWatch Systemprotokollen, die Lambda selbst generiert, an Amazon Logs. Die Ausgabe jeder `console` Methode, z. B., und `console.log` `console.infoconsole.error`, wird automatisch erfasst und anhand der `applicationLogLevel` Einstellung gefiltert. Node.js Protokollbibliotheken, die in die Standardausgabe oder Standardfehlerausgabe schreiben, werden ebenfalls erfasst. Sie benötigen weder einen Logging-Agent noch das Amazon CloudWatch Logs SDK.

Lambda schreibt die `console` Ausgabe als JSON-Protokollereignisse und fügt jedem Ereignis die `region` Felder `requestIdendpointName`, und hinzu. Ein Beispiel finden Sie unter the section called “Strukturierte Logs schreiben”.

Lambda sammelt Protokolle in allen Bereichen, in AWS-Region denen Ihre Webfunktion ausgeführt wird, und zentralisiert sie dann in der Heimatregion der Funktion. Ein `HomeRegion` Endpunkt läuft nur in der Heimatregion. Ein `MultiRegion` `PerRegion` OR-Endpunkt wird auch in den anderen Regionen ausgeführt, die Sie angeben, sodass Lambda die Protokolle aus jeder dieser Regionen zentralisiert. Weitere Informationen finden Sie unter the section called “Endpunkttypen”.

Protokollgruppen

Standardmäßig sendet Lambda die Protokolle Ihrer Webfunktion an eine Protokollgruppe, die nach der Funktion benannt ist:

```
/aws/lambda/web/function-name
```

Neben der zentralisierten Protokollgruppe verwendet Lambda eine zugehörige Protokollgruppe, um regionsinterne Protokolle zu speichern, bevor sie zentralisiert werden. Der Name des

Companions ist der Name der zentralisierten Protokollgruppe `.local`, gefolgt von, sowohl für die Standard-Protokollgruppe als auch für eine benutzerdefinierte Protokollgruppe. Für die Standard-Protokollgruppe lautet der Begleiter:

```
/aws/lambda/web/function-name.local
```

Lambda schreibt Protokolle in die `.local` Protokollgruppe in jeder Region, in der Anfragen ausgeführt werden, einschließlich der Heimatregion, und zentralisiert sie dann in der Protokollgruppe ohne das Suffix in der Heimatregion. Jede `.local` Protokollgruppe enthält nur die Protokolle aus ihrer eigenen Region. Lambda entfernt nach der Zentralisierung keine Ereignisse aus der `.local` Protokollgruppe. Die Aufbewahrungseinstellung der Protokollgruppe bestimmt, wie lange sie verbleiben.

Die Protokollgruppen existieren erst, wenn Ihre Web-Funktion ihre erste Anfrage bearbeitet. Die zentralisierte Protokollgruppe in der Heimatregion wird möglicherweise nach der `.local` Protokollgruppe angezeigt. Informationen zum Zeitpunkt der Zentralisierung finden Sie unterthe section called “Latenz bei der Zentralisierung”.

Lambda erstellt die Standard-Protokollgruppe und jede `.local` Protokollgruppe mit den Standardeinstellungen von Amazon CloudWatch Logs, sodass ihre Ereignisse nie ablaufen, es sei denn, Sie legen eine Aufbewahrungsrichtlinie fest. Sie können ihre Aufbewahrungs-, Verschlüsselungs- und Ressourcenrichtlinien ändern, nachdem Lambda sie erstellt hat. Legen Sie diese in der `.local` Protokollgruppe in jeder Region fest, in der der Endpunkt ausgeführt wird, und in der zentralen Protokollgruppe in der Heimatregion.

Da die `.local` Protokollgruppe nicht auf die Zentralisierung wartet, ist sie nützlich, um ein Problem in einer bestimmten Region zu debuggen. Die zentralisierte Protokollgruppe enthält Protokolle aus allen Regionen. Erstellen Sie daher gemeinsame Abfragen, Abonnementfilter, Metrikfilter und Automatisierungen, die die gesamte Anwendung in der zentralen Protokollgruppe abdecken müssen. Informationen zum Verhalten der Telemetrie in den einzelnen Regionen finden Sie unter. the section called “Telemetrie in jeder Region”

Verwenden einer benutzerdefinierten Protokollgruppe

Sie können die Protokolle Ihrer Webfunktion an eine Protokollgruppe senden, die Sie anstelle der Standardgruppe auswählen. Verwenden Sie eine benutzerdefinierte Protokollgruppe, wenn Sie Richtlinien für Aufbewahrung, Verschlüsselung oder Ressourcen einrichten möchten, bevor das erste Protokollereignis eintrifft. Mit einer benutzerdefinierten Protokollgruppe können Sie auch Protokolle

verschiedener Funktionen gruppieren. Um diese Einstellungen von Anfang an anzuwenden, erstellen Sie die Protokollgruppe und die `.local` zugehörige Gruppe selbst, bevor Sie den Datenverkehr an die Revision weiterleiten. Wenn keine der Protokollgruppen existiert, erstellt Lambda sie bei der ersten Verwendung mit Standardeinstellungen, und Sie können ihre Einstellungen anschließend aktualisieren. Lambda kann eine Protokollgruppe nur erstellen, wenn die Ausführungsrolle über die Berechtigung zum Erstellen von Protokollgruppen verfügt. Die erforderlichen Berechtigungen finden Sie unter the section called “Rolle „Ausführung“”.

Um Protokolle an eine benutzerdefinierte Protokollgruppe zu senden (AWS CLI)

1. Erstellen Sie in der Heimatregion die Protokollgruppe und die `.local` zugehörige Protokollgruppe und legen Sie für jede Gruppe die gewünschte Aufbewahrung fest. Erstellen Sie für einen `MultiRegion PerRegion` OR-Endpunkt auch die `.local` Protokollgruppe in jeder anderen Region, die der Endpunkt bedient, indem Sie die `.local` Befehle mit `--region` set auf diese Region ausführen.

```
aws logs create-log-group --log-group-name /my-app/web-logs
aws logs create-log-group --log-group-name /my-app/web-logs.local

aws logs put-retention-policy \
    --log-group-name /my-app/web-logs \
    --retention-in-days 30

aws logs put-retention-policy \
    --log-group-name /my-app/web-logs.local \
    --retention-in-days 30
```

2. Erstellen Sie eine Revision, die die Protokollgruppe im `telemetryConfig` Abschnitt von spezifiziert`serviceConfig`.

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --build-config '{
      "codeConfig": {
        "s3Object": { "bucket": "my-bucket", "key": "my-app.zip" }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role",
      "telemetryConfig": {
```

```
        "loggingConfig": {
          "logGroup": "/my-app/web-logs"
        }
      }
    }'
```

3. Leiten Sie den Verkehr zur neuen Version weiter. Weitere Informationen finden Sie unter the section called “Bereitstellungsmuster”.

Da die Protokollierungskonfiguration Teil der Revision ist, ist sie für diese Revision unveränderlich. Um die Protokollgruppe später zu ändern, erstellen Sie eine weitere Revision. Weitere Informationen finden Sie unter the section called “Überarbeitungen”.

Kontrollen für die Protokollierung

`loggingConfig`In diesem Abschnitt werden das Ziel und der Mindestschweregrad von Protokollen gesteuert. Diese Einstellungen sind Teil der Revision.

Einstellung	Werte	Standard	Description
`logGroup`	Protokoll gruppenname	`/aws/lambda/web/`*`function-name`*	Die Amazon CloudWatch Logs-Protokollgruppe, die Ihre Protokolle empfängt.
`applicationLogLevel`	`TRACE` | `DEBUG` | `INFO` | `WARN` | `ERROR` | `FATAL`	`INFO`	Der minimale Schweregrad der Protokolle Ihrer Anwendung, die Lambda an Amazon CloudWatch Logs sendet. Protokolle unter dieser Stufe werden gelöscht. Node.jsha t keine `FATAL` Konsolenmethode, daher `FATAL` verhält sich eine Ebene von wie`ERROR`.
`systemLogLevel`	`DEBUG` | `INFO` | `WARN`	`INFO`	Der minimale Schweregrad der Lambda-generated Systemprotokolle, die Lambda an Amazon CloudWatch Logs sendet. Informationen zur Stufe der einzelnen Systemereignisse

Einstellung	Werte	Standard	Description
			finden Sie unterthe section called “Systemprotokolle”.

Im folgenden Beispiel wird eine Revision erstellt, bei der die folgenden Anwendungsprotokolle gelöscht werdenWARN:

```
aws lambda-web create-web-function-revision \
    --function-name my-web-app \
    --build-config '{
      "codeConfig": {
        "s3Object": { "bucket": "my-bucket", "key": "my-app.zip" }
      },
      "runtimeConfig": { "runtime": "nodejs24.x" }
    }' \
    --service-config '{
      "executionRoleArn": "arn:aws:iam::111122223333:role/my-web-function-role",
      "telemetryConfig": {
        "loggingConfig": {
          "applicationLogLevel": "WARN",
          "systemLogLevel": "INFO"
        }
      }
    }'
```

Eine `applicationLogLevel` Erhöhung der Produktion reduziert sowohl das Protokollvolumen als auch die Kosten. Senken Sie den Wert vorübergehend auf `DEBUG` oder `TRACE` wenn Sie ein Problem untersuchen.

Systemprotokolle

Zusätzlich zur Ausgabe Ihrer Anwendung empfängt Ihre Protokollgruppe Systemprotokolle, die Lambda über den Lebenszyklus der Ausführungsumgebung generiert, z. B. die Initialisierung und das Herunterfahren der Umgebung. Wird verwendet`systemLogLevel`, um zu kontrollieren, wie viel von diesem Lambda emittiert wird.

Systemprotokolle sind nützlich, um das Anwendungsverhalten mit Plattformereignissen zu korrelieren. Beispielsweise deutet ein Latenzanstieg, der auf einen Eintrag zur Umgebungsinitialisierung zurückzuführen ist, eher auf einen Kaltstart als auf Ihren Anwendungscode

hin. Informationen zum Lebenszyklus, den diese Protokolle beschreiben, finden Sie unter. the section called “Lebenszyklus der Umgebung”

Webfunktionen geben die folgenden Systemereignisse aus. Stellen Sie `systemLogLevel` die niedrigste Stufe ein, die Sie benötigen. Stellen Sie es beispielsweise so ein`platform.requestComplete`, `DEBUG` dass nur Fehler und Warnungen `WARN` zu gelöschten Protokollen empfangen werden sollen.

Ereignistyp	Level	Hinweise
`platform.initStart`	`INFO`	Keine
`platform.initReport`	`INFO`bei Erfolg, bei Fehlschlag `WARN`	Ein Fehler weist auf einen Initialisierungsfehler hin, z. B. auf einen Fehler beim Importieren Ihres Codes.
`platform.invokeStart`	`INFO`	Keine
`platform.requestComplete`	`DEBUG`	Wird einmal für jede HTTP-Anfrage ausgegeben, einschließlich fehlgeschlagener Anfragen. Da es sich um ein hohes Volumen handelt, wird es nicht auf der Standardprotokollebene des Systems ausgegeben.
`platform.invokeSummary`	`INFO`	Periodisch. Wird etwa alle 60 Sekunden für jede aktive Ausführungsumgebung ausgegeben.
`platform.invokeReport`	`INFO`bei Erfolg, `WARN` bei Misserfolg	
`platform.spinDownStart`	`INFO`	
`platform.spinDownReport`	`INFO`bei Erfolg, `WARN` bei Misserfolg	Ein Fehler weist darauf hin, dass Ihre Anwendung nicht innerhalb von 1

Ereignistyp	Level	Hinweise
		Sekunde nach Empfang von `SIGTERM` (`Runtime.SpinDownTimeout`) beendet wurde oder mit einem Fehler (`Runtime.ExitError`) beendet wurde.
`platform.logsDropped`	`WARN`	Zeigt an, dass Lambda Protokoll ereignisse gelöscht hat.

Informationen zu den Feldern in den einzelnen Systemereignissen finden Sie unterthe section called “Systemprotokollschema”.

Der `lambda-web-request-id` Antwort-Header identifiziert die Anfrage, und der `lambda-web-logstream` Antwort-Header identifiziert den Log-Stream für die Umgebung, in der sie bearbeitet wurde. Verwenden Sie diese Werte, um eine Client-Antwort mit Anwendungs- und Systemprotokollen zu korrelieren. Weitere Informationen finden Sie unter the section called “Antwort-Header”.

Um die Ereignisse aus dem identifizierten Protokollstream in der zentralen Protokollgruppe in der Heimatregion zu finden, filtern Sie nach dem Streamnamen als Präfix. Sie können den Stream auch anhand seines genauen Namens aus der `.local` Protokollgruppe in der Region lesen, die die Anfrage bearbeitet hat.

```
aws logs filter-log-events \
    --log-group-name /aws/lambda/web/my-web-app \
    --log-stream-name-prefix "LAMBDA_WEB_LOGSTREAM" \
    --region us-east-1
```

Um aktuelle Logs zu verfolgen, während ein Problem reproduziert wird, verwende:

```
aws logs tail /aws/lambda/web/my-web-app \
    --follow \
    --region us-east-1
```

Strukturierte Logs schreiben

Übergeben Sie ein Objekt an eine `console` Methode, sodass Lambda es als JSON-Objekt in das `message` Feld schreibt. Sie können dann einzelne Felder in Amazon CloudWatch Logs Insights

abfragen, anstatt sie anhand von Teilzeichenfolgen abzugleichen. Übergeben Sie keine Zeichenfolge, mit `JSON.stringify` der Sie erstellt haben. Lambda schreibt eine Zeichenfolge als maskierten Zeichenkettenwert, sodass ihre Felder nicht abgefragt werden können.

```
function logRequest(req, statusCode, durationMs) {
  console.info({
    method: req.method,
    path: req.url,
    statusCode: statusCode,
    durationMs: durationMs
  });
}
```

Lambda schreibt ein Protokollereignis ähnlich dem folgenden:

```
{
  "timestamp": "2026-09-25T11:29:02.711Z",
  "level": "INFO",
  "message": {
    "method": "GET",
    "path": "/hello-world",
    "statusCode": 200,
    "durationMs": 100
  },
  "requestId": "US3NHLM77IRXDFLBADYHGIZZNM",
  "endpointName": "prod",
  "region": "us-east-1"
}
```

Sie können die Felder dann direkt abfragen. Die folgende Amazon CloudWatch Logs Insights-Abfrage gibt beispielsweise die 20 neuesten Anfragen zurück, die den Status 5xx zurückgegeben haben:

```
fields @timestamp, message.method, message.path, message.statusCode, message.durationMs
| filter message.statusCode >= 500
| sort @timestamp desc
| limit 20
```

Verwenden Sie das eingebettete Metrikformat, um benutzerdefinierte Metriken aus Ihren Protokollen zu veröffentlichen, anstatt sie nur abzufragen. Weitere Informationen finden Sie unter the section called “Benutzerdefinierte Metriken”.

Verwenden von Amazon CloudWatch Logs-Funktionen mit Ihren Protokollen

Da Protokolle in der Heimatregion zentralisiert sind, können Sie die Standardfunktionen von Amazon CloudWatch Logs in dieser Region verwenden, darunter:

- Live Tail — Stream protokolliert Ereignisse, sobald sie eintreffen, während Sie ein Problem reproduzieren. Ereignisse erreichen nach der Zentralisierung die zentrale Protokollgruppe. Um die kürzeste Verzögerung zu erzielen, führen Sie Live Tail für die `.local` Protokollgruppe in der Region aus, die Ihre Anfragen bearbeitet.
- Logs Insights — Abfragen von Feldern in Ihren JSON-Protokollereignissen.
- Zentralisierung von Protokollen — Leiten Sie Ihre Logs mithilfe der Zentralisierungsfunktionen von Amazon CloudWatch Logs an ein anderes Konto oder Ziel weiter, wenn Sie Logs außerhalb des Kontos der Funktion aggregieren.
- Abonnementfilter und Metrikfilter — lösen eine nachgelagerte Verarbeitung aus oder leiten Sie Metriken aus Protokollmustern ab.

Wenden Sie Funktionen, die die gesamte Anwendung abdecken müssen, auf die zentrale Protokollgruppe in der Heimatregion an. Eine `.local` Protokollgruppe enthält nur die Protokolle aus ihrer eigenen Region. Weitere Informationen finden Sie unter the section called “Telemetrie in jeder Region”.

Systemprotokoll-Ereignisschema für Lambda-Webfunktionen

Lambda schreibt die folgenden Systemereignisse in die Protokollgruppe Ihrer Webfunktion. Jedes Ereignis ist ein JSON-Objekt mit den unter beschriebenen Envelope-Feldern the section called “Umschlag für das Ereignis” und einem `record` Objekt, dessen Felder vom Ereignistyp abhängen. Die Stufe jedes Ereignisses bestimmt, ob Lambda es auslöst. Weitere Informationen finden Sie unter the section called “Systemprotokolle”.

Ereignistyp	Objekt aufzeichnen	Description
`platform.initStart`	`PlatformInitStart`	Die Initialisierung der Umgebung wurde gestartet.
`platform.initReport`	`PlatformInitReport`	Bericht über die Initialisierung der Umgebung, unabhängig davon, ob sie erfolgreich war oder fehlgeschlagen ist.

Ereignistyp	Objekt aufzeichnen	Description
`platform.invokeStart`	`PlatformInvokeStart`	Die Umgebung hat begonnen, Anfragen zu bearbeiten.
`platform.requestComplete`	`PlatformRequestComplete`	Eine HTTP-Anfrage wurde abgeschlossen. Nur bei DEBUG protokolliert.
`platform.invokeSummary`	`PlatformInvokeSummary`	Zusammenfassung der Umgebungsaktivität, die etwa alle 60 Sekunden gesendet wird.
`platform.invokeReport`	`PlatformInvokeReport`	Bericht über den gesamten Servierzeitraum der Umgebung, der ausgegeben wird, wenn der Dienst eingestellt wird.
`platform.spinDownStart`	`PlatformSpinDownStart`	Das Herunterfahren der Umgebung wurde gestartet.
`platform.spinDownReport`	`PlatformSpinDownReport`	Bericht über das Herunterfahren der Umgebung.
`platform.logsDropped`	`PlatformLogsDropped`	Protokolldatensätze wurden gelöscht, weil Protokolle schneller erstellt wurden, als sie übermittelt werden konnten.

In den Feldlisten auf dieser Seite bedeutet ein Fragezeichen (?) hinter einem Feldnamen, dass das Feld möglicherweise nicht vorhanden ist. Typen, bei denen es sich nicht um JSON-Typen handelt `InitTypeStatus`, wie`InvokeSummaryMetrics`, oder, sind in definiertthe section called “Freigegebene Objekttypen”, und jeder Typ ist mit seiner Definition verknüpft.

`Platform.InitStart`

Die Initialisierungsphase der Umgebung wurde gestartet.

Stufe: `INFO`

Das `record`-Objekt enthält folgende Felder:

- `functionName`(`String`) — Der Name der Funktion.

- `revisionId`(`String`) — Die Revision, die die Umgebung ausführt.
- `instanceId`(`String`) — Der Bezeichner der Ausführungsumgebung.
- `initializationType`(`InitType`) — Wie Lambda die Umgebung initialisiert hat.
- `runtimeVersion`(`String`) — Die Laufzeitversion, die in der Umgebung ausgeführt wird.
- `runtimeVersionArn`(`String`) — Der Amazon-Ressourcenname (ARN) der Runtime-Version.

```
{
  "timestamp": "2026-09-25T11:12:30.893Z",
  "level": "INFO",
  "type": "platform.initStart",
  "region": "us-east-1",
  "record": {
    "runtimeVersion": "nodejs24.x:nodejs.web24.v51",
    "runtimeVersionArn": "arn:aws:lambda:us-east-1::runtime/nodejs24.x/version/
nodejs.web24.v51",
    "functionName": "my-web-app",
    "revisionId": "rev-fs0fq",
    "instanceId": "52754708-44f1-407a-a1fd-6d2a98175a45",
    "initializationType": "native-http"
  }
}
```

`Platform.initReport`

Gesamtbericht über die Initialisierungsphase.

Stufe: `INFO` wenn der Status Erfolg ist, `WARN` wenn Fehler.

Das `record`-Objekt enthält folgende Felder:

- `status`(`Status`) — Das Ergebnis der Initialisierungsphase.
- `errorType?`(`String`) — Vorhanden, wenn der Status „Fehler“ lautet. Werte wie `Runtime.ImportCodeErrorApplication.InitCodeError`, und`Runtime.InitTimeout`.
- `initializationType`(`InitType`) — Wie Lambda die Umgebung initialisiert hat.
- `metrics`(`InitReportMetrics`) — Metriken für die Initialisierungsphase.

Beispiel für einen Erfolg:

```
{
  "timestamp": "2026-09-25T11:12:31.143Z",
  "level": "INFO",
  "type": "platform.initReport",
  "region": "us-east-1",
  "record": {
    "status": "success",
    "initializationType": "native-http",
    "metrics": {
      "totalDurationMs": 304.000,
      "totalCpuTimeMs": 143.000,
      "maxMemoryUsageMb": 142
    }
  }
}
```

Beispiel für einen Fehler, bei dem das Handler-Modul nicht geladen werden konnte:

```
{
  "timestamp": "2026-09-25T11:16:04.164Z",
  "level": "WARN",
  "type": "platform.initReport",
  "region": "us-east-1",
  "record": {
    "status": "failure",
    "errorType": "Runtime.ImportCodeError",
    "initializationType": "native-http",
    "metrics": {
      "totalDurationMs": 320.000,
      "totalCpuTimeMs": 137.000,
      "maxMemoryUsageMb": 138
    }
  }
}
```

`Platform.invokeStart`

Die Umgebung hat die Initialisierung abgeschlossen und mit der Bearbeitung von Anfragen begonnen. Lambda gibt dieses Ereignis einmal pro Umgebung aus, nicht einmal pro Anfrage, und nur dann, wenn die Initialisierung erfolgreich war.

Stufe: `INFO`

Das `record`-Objekt enthält folgende Felder:

- `functionName`(`String`) — Der Name der Funktion.
- `revisionId`(`String`) — Die Revision, die die Umgebung ausführt.
- `instanceId`(`String`) — Der Bezeichner der Ausführungsumgebung.

```
{
  "timestamp": "2026-09-25T11:12:31.144Z",
  "level": "INFO",
  "type": "platform.invokeStart",
  "region": "us-east-1",
  "record": {
    "functionName": "my-web-app",
    "revisionId": "rev-fs0fq",
    "instanceId": "52754708-44f1-407a-a1fd-6d2a98175a45"
  }
}
```

`Platform.RequestComplete`

Eine HTTP-Anfrage wurde abgeschlossen, erfolgreich oder nicht.

Level:`DEBUG`, Lambda liefert es also nur dann, wenn es `systemLogLevel` ist`DEBUG`.

Das `record`-Objekt enthält folgende Felder:

- `requestId`(`String`) — Die Anforderungs-ID.
- `endpointName`(`String`) — Der Endpunkt, der die Anfrage bedient hat.
- `result`(`Integer`) — Der an den Client zurückgegebene HTTP-Statuscode.
- `errorType?`(`String`) — Anwesend, wenn Lambda die Anfrage nicht bestanden hat. Werte wie `Runtime.RequestTimeout` und`Platform.InvalidRequest`.
- `metrics`(`RequestCompleteMetrics`) — Metriken für die Anfrage.

Beispiel für einen Erfolg:

```
{
  "timestamp": "2026-09-25T11:12:31.154Z",
  "level": "DEBUG",
  "type": "platform.requestComplete",
```

```
  "region": "us-east-1",
  "record": {
    "requestId": "KXH25FBDZ33GP4TQ4ARAUYSTSA",
    "endpointName": "default",
    "result": 200,
    "metrics": {
      "durationMs": 4.484,
      "responseBytes": 40
    }
  }
}
```

Beispiel für ein Timeout:

```
{
  "timestamp": "2026-09-25T11:12:34.926Z",
  "level": "DEBUG",
  "type": "platform.requestComplete",
  "region": "us-east-1",
  "record": {
    "requestId": "5SXHGVX2GD64ND6NQVEBAY235Q",
    "endpointName": "default",
    "result": 504,
    "errorType": "Runtime.RequestTimeout",
    "metrics": {
      "durationMs": 3001.229,
      "responseBytes": 0
    }
  }
}
```

`Platform.InvokeSummary`

Fasst die Aktivität einer Umgebung in einem Fenster von etwa 60 Sekunden zusammen. Lambda gibt es für jede Live-Umgebung aus, einschließlich inaktiver Fenster, wo 0 `requests` ist.

Stufe: `INFO`

Das `record`-Objekt enthält folgende Felder:

- `startTime`(`String`) — Der Beginn des Übersichtsfensters (RFC 3339).
- `instanceId`(`String`) — Der Bezeichner der Ausführungsumgebung.

- `metrics`(`InvokeSummaryMetrics`) — Metriken für das Fenster.

```
{
  "timestamp": "2026-09-25T11:13:31.143Z",
  "level": "INFO",
  "type": "platform.invokeSummary",
  "region": "us-east-1",
  "record": {
    "startTime": "2026-09-25T11:12:31.144547426Z",
    "instanceId": "52754708-44f1-407a-a1fd-6d2a98175a45",
    "metrics": {
      "requests": 4,
      "concurrentRequests": 1,
      "cpuTimeMs": 420.000,
      "maxMemoryUsageMb": 154
    }
  }
}
```

`Platform.InvokeReport`

Gesamtbericht über den Bereitstellungszeitraum der Umgebung, von invokeStart bis zum Ende der Bereitstellung. Lambda gibt es einmal pro Umgebung aus, beim Herunterfahren.

Stufe: `INFO` wenn der Status Erfolg ist, `WARN` wenn Fehler.

Das `record`-Objekt enthält folgende Felder:

- `status`(`Status`) — Das Ergebnis der Dienstzeit.
- `errorType?`(`String`) — Vorhanden, wenn der Status „Fehlgeschlagen“ lautet. Werte wie `Runtime.ExitRuntime.ExitError`, und`Runtime.OutOfMemory`.
- `instanceId`(`String`) — Der Bezeichner der Ausführungsumgebung.
- `metrics`(`InvokeReportMetrics`) — Kennzahlen für den Dienstzeitraum.

Bei einem Zeitlimit für Anfragen wird der Bereitstellungszeitraum nicht überschritten. Die `status` Aufenthaltes`success`.

Beispiel für einen Erfolg:

```
{
```

```
  "timestamp": "2026-09-25T11:19:11.858Z",
  "level": "INFO",
  "type": "platform.invokeReport",
  "region": "us-east-1",
  "record": {
    "status": "success",
    "instanceId": "52754708-44f1-407a-a1fd-6d2a98175a45",
    "metrics": {
      "totalCpuTimeMs": 973.000,
      "totalDurationMs": 400723.000,
      "totalRequests": 4,
      "concurrentRequests": 1,
      "maxMemoryUsageMb": 157
    }
  }
}
```

Beispiel für einen Misserfolg:

```
{
  "timestamp": "2026-09-25T11:19:11.858Z",
  "level": "WARN",
  "type": "platform.invokeReport",
  "region": "us-east-1",
  "record": {
    "status": "failure",
    "errorType": "Runtime.OutOfMemory",
    "instanceId": "52754708-44f1-407a-a1fd-6d2a98175a45",
    "metrics": {
      "totalCpuTimeMs": 973.000,
      "totalDurationMs": 400723.000,
      "totalRequests": 4,
      "concurrentRequests": 1,
      "maxMemoryUsageMb": 512
    }
  }
}
```

`platform.spin DownStart`

Das Herunterfahren der Umgebung wurde gestartet.

Stufe: `INFO`

Das `record`-Objekt enthält folgende Felder:

- `functionName`(`String`) — Der Name der Funktion.
- `revisionId`(`String`) — Die Revision, die die Umgebung ausführt.
- `instanceId`(`String`) — Der Bezeichner der Ausführungsumgebung.

```
{
  "timestamp": "2026-09-25T11:19:11.858Z",
  "level": "INFO",
  "type": "platform.spinDownStart",
  "region": "us-east-1",
  "record": {
    "functionName": "my-web-app",
    "revisionId": "rev-fs0fq",
    "instanceId": "52754708-44f1-407a-a1fd-6d2a98175a45"
  }
}
```

`platform.spin DownReport`

Gesamtbericht über den Shutdown.

Lambda sendet `SIGTERM` an Ihre Anwendung und wartet bis zu 1 Sekunde, bis sie beendet wird. Wenn die Anwendung noch läuft, stoppt Lambda sie `SIGKILL` und meldet einen Fehler mit dem Fehlertyp`Runtime.SpinDownTimeout`.

Stufe: `INFO` wenn der Status Erfolg ist, `WARN` wenn Fehler.

Das `record`-Objekt enthält folgende Felder:

- `status`(`Status`) — Das Ergebnis der Shutdown-Phase.
- `errorType?`(`String`) — Vorhanden, wenn der Status „Fehler“ lautet. Werte wie `Runtime.SpinDownTimeout` und`Runtime.ExitError`.
- `metrics`(`SpinDownReportMetrics`) — Metriken für die Shutdown-Phase.

Beispiel für einen Erfolg:

```
{
```

```
  "timestamp": "2026-09-25T11:19:11.879Z",
  "level": "INFO",
  "type": "platform.spinDownReport",
  "region": "us-east-1",
  "record": {
    "status": "success",
    "metrics": {
      "totalDurationMs": 21.000,
      "totalCpuTimeMs": 8.000,
      "maxMemoryUsageMb": 157
    }
  }
}
```

Beispiel für einen Misserfolg:

```
{
  "timestamp": "2026-09-25T11:19:12.879Z",
  "level": "WARN",
  "type": "platform.spinDownReport",
  "region": "us-east-1",
  "record": {
    "status": "failure",
    "errorType": "Runtime.SpinDownTimeout",
    "metrics": {
      "totalDurationMs": 1021.000,
      "totalCpuTimeMs": 8.000,
      "maxMemoryUsageMb": 157
    }
  }
}
```

`Platform.Logs wurde gelöscht`

Protokolldatensätze wurden gelöscht, weil Protokolle schneller erstellt wurden, als sie übermittelt werden konnten.

Stufe: `WARN`

Das `record`-Objekt enthält folgende Felder:

- `reason`(`String`) — Warum die Aufzeichnungen gelöscht wurden.

- `droppedRecords`(`Integer`) — Datensätze, die seit dem letzten gelieferten Datensatz gelöscht wurden.
- `droppedBytes`(`Integer`) — Seit dem letzten Datensatz, der geliefert wurde, verloren gegangene Byte.

Beide Zählungen summieren sich bei aufeinanderfolgenden Drops und werden nach der Lieferung eines Datensatzes auf 0 zurückgesetzt.

```
{
  "timestamp": "2026-09-25T11:20:00.000Z",
  "level": "WARN",
  "type": "platform.logsDropped",
  "region": "us-east-1",
  "record": {
    "reason": "Some logs were dropped because the downstream consumer is slower than
 the logs production rate",
    "droppedRecords": 120,
    "droppedBytes": 48000
  }
}
```

Freigegebene Objekttypen

Die folgenden Typen werden in den Feldlisten der Ereignisse auf dieser Seite angezeigt.

Umschlag für das Ereignis

Jedes Systemereignis hat die folgenden Envelope-Felder:

- `timestamp`(`String`) — RFC 3339 in UTC.
- `level`(`Level`) — Die Ebene des Ereignisses.
- `type`(`String`) — Der Ereignistyp (`platform.*`).
- `region`(`String`) — Die Region, in der die Ausführungsumgebung lief.
- `record`(`Object`) — Die ereignisspezifischen Felder.

`InitType`

String-Aufzählung, die beschreibt, wie Lambda die Umgebung initialisiert hat. Webfunktionen verwenden immer. `native-http`

`Level`

Zeichenkettenaufzählung für die Ebene eines Systemereignisses:`DEBUG`,`INFO`, oder`WARN`. Systemereignisse sind niemals `ERROR` oder`FATAL`. Berichtsereignisse liegen vor, `INFO` wenn der Status erfolgreich und `WARN` der Status fehlgeschlagen ist.

`Status`

Zeichenkettenaufzählung, die das Ergebnis einer Phase beschreibt: `success` oder`failure`. Wenn ja`failure`, enthält `errorType` der Datensatz auch.

`InitReportMetrics`

- `totalDurationMs`(`Double`) — Wall-clock Zeitpunkt der Initialisierungsphase.
- `totalCpuTimeMs`(`Double`) — Während der Initialisierung verbrauchte CPU-Zeit.
- `maxMemoryUsageMb`(`Integer`) — Spitzenspeicher der Umgebung, in MiB.

`RequestCompleteMetrics`

- `durationMs`(`Double`) — Wall-clock Uhrzeit der Anfrage.
- `responseBytes`(`Integer`) — Byte des gesendeten Antworttextes.

`InvokeSummaryMetrics`

- `requests`(`Integer`) — Im Fenster abgeschlossene Anfragen.
- `concurrentRequests`(`Integer`) — Maximale Anzahl gleichzeitiger Anfragen im Fenster.
- `cpuTimeMs`(`Double`) — Im Fenster verbrauchte CPU-Zeit.
- `maxMemoryUsageMb`(`Integer`) — Spitzenspeicher der Umgebung, in MiB.

`InvokeReportMetrics`

- `totalCpuTimeMs`(`Double`) — CPU-Zeit während des Bereitstellungszeitraums.
- `totalDurationMs`(`Double`) — Wall-clock Zeit von invokeStart bis zum Ende der Serverbereitstellung in der Umgebung.
- `totalRequests`(`Integer`) — Gesamtzahl der bearbeiteten Anfragen.
- `concurrentRequests`(`Integer`) — Maximale Anzahl gleichzeitiger Anfragen im Laufe des Zeitraums.

- `maxMemoryUsageMb(Integer)` — Spitzenspeicher der Umgebung, in MiB.

`SpinDownReportMetrics`

- `totalDurationMs(Double)` — Wall-clock Zeitpunkt der Abschaltphase.
- `totalCpuTimeMs(Double)` — Beim Herunterfahren verbrauchte CPU-Zeit.
- `maxMemoryUsageMb(Integer)` — Spitzenspeicher der Umgebung, in MiB.

Veröffentlichen von benutzerdefinierten Metriken mit dem eingebetteten Metrikformat

Mit dem Amazon CloudWatch Embedded Metric Format (EMF) können Sie Ihre eigenen Kennzahlen aus AWS Lambda Web Functions veröffentlichen, z. B. aufgegebene Bestellungen, Cache-Trefferquote oder Zahlungslatenz. Ihre Anwendung schreibt ein strukturiertes JSON-Objekt in ihre Protokolle und CloudWatch extrahiert die Metriken daraus. Verwenden Sie benutzerdefinierte Metriken, um zu verfolgen, was Ihre Anwendung tut. Die von Lambda veröffentlichten Metriken beschreiben nur die Ergebnisse von Anfragen und die Ressourcennutzung.

Sie können auch benutzerdefinierte Metriken veröffentlichen, indem Sie die CloudWatch `PutMetricData` API direkt aufrufen. Wir empfehlen jedoch EMF für Webfunktionen:

- Keine zusätzliche Latenz bei Anfragen — Ihre Anwendung ruft die CloudWatch API nicht auf dem Pfad auf, der Ihre HTTP-Antwort bereitstellt.
- Keine zusätzliche Erlaubnis — Sie benötigen die `cloudwatch:PutMetricData` Erlaubnis nicht. EMF verwendet dieselben Amazon CloudWatch Logs-Berechtigungen, die die Ausführungsrolle bereits für Protokolle benötigt.
- Zentralisierung Ihrer Logs — Da EMF-Metriken zusammen mit Ihren Logs übertragen werden, enthält die Heimatregion Werte aus allen Regionen, die der Endpunkt bedient. Metriken, mit denen Sie veröffentlichen, `PutMetricData` werden nur in der Region angezeigt, in der Sie sie aufrufen.

So funktioniert EMF mit Webfunktionen

EMF-Metriken werden von Ihrer Anwendung zu den folgenden Schritten CloudWatch weitergeleitet:

1. Ihre Anwendung schreibt ein EMF-formatted JSON-Objekt, gefolgt von einem Zeilenumbruch, direkt in die Standardausgabe.

2. Lambda erfasst es als Protokollereignis in der Region, in der die Anfrage ausgeführt wurde AWS-Region, und zentralisiert es wie jede andere Protokollausgabe im Home. Details hierzu finden Sie unter the section called “Protokolle”.
3. CloudWatch extrahiert Metriken aus dem Objekt in der Region, in der die Anfrage ausgeführt wurde, und Lambda repliziert die Metriken in die Heimatregion. Die Metriken werden in dem Namespace angezeigt, den Sie im Objekt angeben. In der Heimatregion enthalten die Metriken Werte aus jeder Region, die der Endpunkt bedient.

Ihre benutzerdefinierten Metriken werden in Ihrem eigenen Namespace angezeigt, nicht in`AWS/Lambda/Web`. Dieser Namespace ist für die von Lambda veröffentlichten Metriken reserviert.

Veröffentlichung einer benutzerdefinierten Metrik

Im folgenden Node.js Beispiel werden eine `CheckoutLatency` Metrik und eine `OrdersPlaced` Metrik unter dem `MyWebApp` Namespace mit einer `PaymentProvider` Dimension veröffentlicht. Es schreibt das EMF-Objekt mit. `process.stdout.write` Übergeben Sie kein EMF-Objekt an eine `console` Methode. Lambda verpackt die `console` Ausgabe in ein eigenes JSON-Protokollereignis und extrahiert CloudWatch keine Metriken aus dem verpackten Objekt.

```
function publishCheckoutMetrics(provider, latencyMs, orders) {
  process.stdout.write(JSON.stringify({
    _aws: {
      Timestamp: Date.now(),
      CloudWatchMetrics: [
        {
          Namespace: 'MyWebApp',
          Dimensions: [['PaymentProvider']],
          Metrics: [
            { Name: 'CheckoutLatency', Unit: 'Milliseconds' },
            { Name: 'OrdersPlaced', Unit: 'Count' }
          ]
        }
      ]
    },
    PaymentProvider: provider,
    CheckoutLatency: latencyMs,
    OrdersPlaced: orders,
    // Fields outside the metric definitions are kept as searchable log context.
    requestPath: '/checkout'
  }) + '\n');
```

```
}
```

Anstatt EMF-Objekte selbst zu erstellen, können Sie eine EMF-Clientbibliothek wie aws-embedded-metrics für die Website verwenden. Node.js GitHub Eine Liste der Bibliotheken finden Sie unter Erstellen von Protokollen im eingebetteten metrischen Format mithilfe der Client-Bibliotheken im CloudWatch Amazon-Benutzerhandbuch. Bevor Sie sich auf eine Bibliothek verlassen, sollten Sie sicherstellen, dass sie EMF-Objekte direkt in die Standardausgabe einer Webfunktion schreibt und nicht über `console` oder an einen CloudWatch Agenten.

Beachten Sie die drei Rollen, die Felder in einem EMF-Objekt spielen:

- Felder mit den Namen `Metrics` werden zu CloudWatch Metriken.
- In benannte Felder `Dimensions` werden zu metrischen Dimensionen. Halten Sie die Anzahl der unterschiedlichen Dimensionswerte gering. Jede Kombination erzeugt eine separate Metrik.
- Jedes andere Feld wird im Protokollereignis als Kontext gespeichert, den Sie mit Amazon CloudWatch Logs Insights abfragen können, erstellt aber keine Metrik.

Verwenden Sie die letzte Eigenschaft bewusst: Fügen Sie Werte mit hoher Kardinalität wie eine Anforderungs- oder Kundennummer als einfache Felder und nicht als Dimensionen hinzu, sodass Sie die spezifische Anfrage in Logs Insights finden können, ohne die Anzahl Ihrer Metriken zu multiplizieren.

Abrechnung für benutzerdefinierte Metriken

EMF generiert sowohl Gebühren für die Protokollaufnahme als auch für benutzerdefinierte Metriken. Das Protokollereignis wird zu Amazon CloudWatch Logs-Preisen abgerechnet, und die extrahierten Metriken werden als CloudWatch benutzerdefinierte Metriken abgerechnet. Die erste Zentralisierung eines Protokollereignisses ist kostenlos. CloudWatch extrahiert EMF-Metriken in der Region, in der die Anfrage ausgeführt wurde, und Lambda repliziert sie in die Heimatregion, sodass eine Anfrage, die außerhalb der Heimatregion bedient wird, benutzerdefinierte Metriken in beiden Regionen erzeugt.

Das Protokollvolumen, die Anzahl der Metriken und die Anzahl der unterschiedlichen Dimensionskombinationen wirken sich auf die Kosten aus. Veröffentlichen Sie Metriken mit der Granularität, für die Sie einen Alarm ausgeben oder grafisch darstellen möchten, und vermeiden Sie Dimensionen mit hoher Kardinalität.

Aktuelle Preise finden Sie unter CloudWatchAmazon-Preise.

Protokollieren von API-Aufrufen für Webfunktionen mit AWS CloudTrail

Mit können Sie die Aktionen verfolgen AWS CloudTrail, die von einem Benutzer, einer Rolle oder AWS-Service in AWS Lambda Web Functions ausgeführt wurden. CloudTrail bietet eine Aufzeichnung dieser Aktionen. CloudTrail erfasst Aufrufe von Webfunktionen als Datenereignisse. Sie können diese Informationen verwenden, um festzustellen, wer wann und von wo aus eine Webfunktion aufgerufen hat, was bei der Sicherheitsanalyse und Betriebsprüfung hilfreich ist.

Datenereignisse der Webfunktion in CloudTrail

Datenereignisse liefern Informationen über Operationen, die an einer Ressource ausgeführt werden. Beispielsweise ist das Aufrufen eines Webfunktionsendpunkts ein Datenereignis. Diese Operationen werden auch als Operationen auf Datenebene bezeichnet und sind häufig sehr umfangreich. Standardmäßig werden CloudTrail keine Datenereignisse protokolliert, und im CloudTrail Ereignisverlauf werden sie nicht aufgezeichnet.

Für Datenereignisse werden zusätzliche Gebühren fällig. Weitere Informationen finden Sie unter CloudTrail – Preise.

Sie können Datenereignisse für den `AWS::Lambda::WebFunction` Ressourcentyp mithilfe der CloudTrail Konsole oder CloudTrail API-Operationen protokollieren. AWS CLI CloudTrail zeichnet jeden Aufruf in der Region auf AWS-Region , die die Anfrage bedient hat, und `awsRegion` identifiziert diese Region. Im Gegensatz zu Protokollen und Metriken werden Aufrufereignisse nicht in der Heimatregion zentralisiert. Verwenden Sie für einen `MultiRegion` `PerRegion` OR-Endpunkt einen Trail- oder Event-Datenspeicher mit mehreren Regionen, sodass Aufrufe in jeder Region, die der Endpunkt bedient, aufgezeichnet werden.

Die folgende Tabelle zeigt den Datenereignistyp für Webfunktionen.

Typ des Datenereignisses (Konsole)	resources.type-Wert	Daten-APIs, bei denen Sie angemeldet sind CloudTrail
Lambda-Webfunktion	`AWS::Lambda::WebFunction`	`InvokeWebFunctionEndpoint`

CloudTrail zeichnet die folgenden Schlüsselbezeichner für den Aufruf einer Webfunktion auf:

Kennung	Wert
`eventSource`	`lambda.amazonaws.com`
`eventName`	`InvokeWebFunctionEndpoint` . Dies ist der Wert, nach dem gefiltert werden soll.
IAM-Aktion	`lambda:InvokeWebFunctionEndpoint`
`readOnly`	`false`, weil ein Aufruf Ihren Funktionscode ausführt.
`resources.type`	`AWS::Lambda::WebFunction`
`resources.ARN`	Die Funktion ARN, nicht der Endpunkt-ARN. Trail-Selektoren funktionieren daher auf Funktionsebene.

Der folgende erweiterte Event-Selektor protokolliert Aufrufe nur für eine einzelne Webfunktion.

```
[
    {
        "Name": "web-function-invokes",
        "FieldSelectors": [
            { "Field": "eventCategory", "Equals": ["Data"] },
            { "Field": "resources.type", "Equals": ["AWS::Lambda::WebFunction"] },
            { "Field": "resources.ARN", "Equals": ["arn:aws:lambda:us-
east-1:111122223333:web-function/hello-world"] }
        ]
    }
]
```

Gehen Sie wie folgt vor, um Datenereignisse auf einem Trail zu aktivieren.

Um Datenereignisse der Webfunktion zu aktivieren und zu überprüfen (AWS CLI)

1. Erstellen Sie den Trail, der die Datenereignisse aufzeichnet, oder wählen Sie ihn aus. Anweisungen finden Sie im AWS CloudTrail Benutzerhandbuch unter Protokollieren von Datenereignissen.

2. Speichern Sie den folgenden erweiterten Event-Selektor unter. `web-function-data-events.json` Er zeichnet Aufrufe für eine Webfunktion auf. Um Aufrufe für jede Webfunktion im Konto aufzuzeichnen, entfernen Sie die Feldauswahl. `resources.ARN`

```
[
  {
    "Name": "web-function-invocations",
    "FieldSelectors": [
      {
        "Field": "eventCategory",
        "Equals": ["Data"]
      },
      {
        "Field": "resources.type",
        "Equals": ["AWS::Lambda::WebFunction"]
      },
      {
        "Field": "resources.ARN",
        "Equals": [
          "arn:aws:lambda:us-east-1:111122223333:web-function/my-web-app"
        ]
      }
    ]
  }
]
```

3. Wenden Sie den Selektor auf den Trail an:

```
aws cloudtrail put-event-selectors \
    --trail-name my-trail \
    --advanced-event-selectors file://web-function-data-events.json
```

> **Important**
>
> `put-event-selectors`ersetzt die vorhandenen Event-Selektoren des Trails. Wenn der Trail bereits andere Ereignisse aufzeichnet, fügen Sie deren Selektoren in dieselbe Datei ein.

4. Rufen Sie den Endpunkt mit einer Testanforderung auf.
5. Fragen Sie das Ziel des Pfads für `eventName = InvokeWebFunctionEndpoint` ab. Überprüfen Sie den Funktions-ARN, die Ereigniszeit, die Anforderungs-ID, die für den

Authentifizierungspfad verfügbaren Anruferinformationen und alle Felder für bedingte Fehler. Wenn der Selektor funktioniert, gibt die Abfrage ein `InvokeWebFunctionEndpoint` Ereignis für Ihre Testanforderung zurück, ähnlich dem Beispiel unter. the section called “Beispiel: erfolgreicher Aufruf (nicht authentifizierter Anrufer)”

Um Datenereignisse stattdessen in einem CloudTrail Lake-Ereignisdatenspeicher aufzuzeichnen, fügen Sie dem Ereignisdatenspeicher dieselben Feldselektoren hinzu. Weitere Informationen finden Sie im AWS CloudTrail -Benutzerhandbuch.

Felder in einem Ereignis InvokeWebFunctionEndpoint

In der folgenden Tabelle werden die Felder in einem `InvokeWebFunctionEndpoint` Ereignis beschrieben.

Feld	Hinweise
`eventVersion`	Die Version des -Ereignisformats.
`eventSource`	Immer `lambda.amazonaws.com` .
`eventName`	Immer `InvokeWebFunctionEndpoint` .
`eventCategory`	Immer `Data`.
`eventType`	Zum Beispiel der Ereignistyp`AwsApiCall` .
`readOnly`	Immer`false`, weil ein Aufruf Ihren Funktionscode ausführt.
`managementEvent`	Immer `false`.
`eventTime`	Datum und Uhrzeit in UTC, an dem Lambda das Ereignis aufgezeichnet hat. Für einen erfolgreichen Aufruf zeichnet Lambda das Ereignis auf, wenn Ihre Funktion ihre Antwort startet, also `eventTime` kann es später als zu dem Zeitpunkt sein, zu dem die Anfrage eingetroffen ist.
`sourceIPAddress`	Die IP-Adresse, von der die Anforderung erfolgt ist.

Feld	Hinweise
`userAgent`	Der Agent, mit dem der Anrufer die Anfrage gestellt hat. Anwesend, wenn der Anrufer einen `User-Agent` Header sendet.
`requestParameters`	Enthält `endpointArn` den ARN des aufgerufenen Endpunkts. Die Payload der Anfrage wird nicht protokolliert.
`responseElements`	Immer `null`.
`resources`	Enthält einen Eintrag mit den `accountId` `type`, und ARN der Funktion.
`additionalEventData`	Enthält `functionRevisionArn` den ARN der Revision, die die Anfrage bearbeitet hat. Dieses Feld ist bei jedem Ereignis vorhanden, auch bei Ablehnungen.
`tlsDetails`	Die TLS-Version, die Cipher Suite und der Host-Header der Anfrage.
`userIdentity`	Die Identität des Anrufers. Einzelheiten finden Sie in der folgenden Liste.
`sharedEventID`	Vorhanden CloudTrail , wenn dasselbe Ereignis an mehrere Personen gesendet wird AWS-Konto.
`errorCode` , `errorMessage`	Nur bei einem Fehlerereignis vorhanden.
`requestID` , `eventID`, `recipientAccountId` , `awsRegion`	CloudTrail Standardfelder, die die Anfrage, das Ereignis, den Empfänger AWS-Konto und den identifizieren AWS-Region.

Web-Funktionsereignisse verwenden die folgenden ARN-Formen:

- Funktion (`resources[].ARN`) — `arn:aws:lambda:us-east-1:123456789012:web-function/my-fn`

- Endpunkt (`requestParameters.endpointArn`) — `arn:aws:lambda:us-east-1:123456789012:web-function/my-fn/endpoint/my-endpoint`
- Revision (`additionalEventData.functionRevisionArn`) — `arn:aws:lambda:us-east-1:123456789012:web-function/my-fn/revision/rev-a1b2c`

Das `userIdentity` Feld hängt davon ab, wie sich der Anrufer authentifiziert hat:

- Nicht authentifizierte Aufrufe werden mit Typ aufgezeichnet. `userIdentity` `Unknown`
- IAM-authenticated Anrufer werden mit ihrem eigenen Principal protokolliert. Beispielsweise wird ein Anrufer, der temporäre Anmeldeinformationen verwendet`AssumedRole`, mit den Typen an `arn` und a angezeigt. `sessionContext`
- Das Feld `accessKeyId` wird nicht gefüllt.

Fehlerereignisse

CloudTrail protokolliert auch Aufrufversuche, die fehlschlagen, nachdem der Anrufer authentifiziert wurde. In der folgenden Tabelle werden diese Fehlerereignisse beschrieben.

Szenario	errorCode	errorMessage
Der Anrufer ist nicht autorisiert	`AccessDenied`	Der Benutzer ist nicht berechtigt, Folgendes auszuführen: `lambda:InvokeWebFunctionEndpoint`
Anforderungsrate, Parallelität oder vCPU-Limit erreicht (Endpunkt oder Konto)	`ThrottlingException`	Quote überschritten.
Dienstfehler	`ServiceException`	Servicefehler
Anwendungsfehler oder ungültige Client-Anfrage	Nicht zutreffend	Wird als erfolgreicher API-Aufruf behandelt. Der Aufruf hat die Funktion erreicht und ausgeführt.

Bei einem erfolgreichen Aufruf gibt es kein Oder. `errorCode` `errorMessage` Autorisierungsverweigerungen werden als protokolliert. `AccessDenied` CloudTrail zeichnet keine Anfragen auf, bei denen die Authentifizierung fehlschlägt, wie Anfragen mit einer fehlenden, falsch formatierten, ungültigen oder abgelaufenen Signatur, und zeichnet keine Anfragen auf, die Lambda vor der Authentifizierung ablehnt, wie Anfragen an einen Endpunkt, dessen `rateLimit` Wert 0 ist. Ausfälle der internen Authentifizierungsinfrastruktur werden ebenfalls nicht aufgezeichnet. Um einen Anstieg der abgelehnten Anfragen zu erkennen, aktivieren Sie den Alarm `HTTPResponseStatusClass4XX` im `AWS/Lambda/Web` Namespace.

Beispiel: erfolgreicher Aufruf (nicht authentifizierter Anrufer)

Das folgende Beispiel zeigt einen erfolgreichen Aufruf durch einen nicht authentifizierten Anrufer.


```
{
    "Records": [
        {
            "eventVersion": "1.11",
            "userIdentity": {
                "type": "Unknown"
            },
            "eventTime": "2026-09-22T18:41:07Z",
            "eventSource": "lambda.amazonaws.com",
            "eventName": "InvokeWebFunctionEndpoint",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "203.0.113.42",
            "userAgent": "curl/8.4.0",
            "requestParameters": {
                "endpointArn": "arn:aws:lambda:us-east-1:123456789012:web-function/my-
fn/endpoint/my-endpoint"
            },
            "responseElements": null,
            "additionalEventData": {
                "functionRevisionArn": "arn:aws:lambda:us-east-1:123456789012:web-
function/my-fn/revision/rev-a1b2c"
            },
            "requestID": "b1f8a4c2-7e3d-4a19-9f6b-2c5d8e0a1234",
            "eventID": "d4c3b2a1-5f6e-7a8b-9c0d-1e2f3a4b5c6d",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "123456789012",
                    "type": "AWS::Lambda::WebFunction",
```

```
                    "ARN": "arn:aws:lambda:us-east-1:123456789012:web-function/my-fn"
                }
            ],
            "eventType": "AwsApiCall",
            "managementEvent": false,
            "recipientAccountId": "123456789012",
            "eventCategory": "Data",
            "tlsDetails": {
                "tlsVersion": "TLSv1.3",
                "cipherSuite": "TLS_AES_128_GCM_SHA256",
                "clientProvidedHostHeader": "bcdfghjklmnpqrst.b7t.lambda-web.us-
east-1.on.aws"
            }
        }
    ]
}
```

Bei einem IAM-authenticated Anrufer enthält das `userIdentity` Feld stattdessen den Principal des Aufrufers, z. B. Typ `AssumedRole` mit einem `arn` und einem`sessionContext`, und die übrigen Felder bleiben unverändert.

Beispiel: Abgelehnter Aufruf

Fehlerereignisse werden übertragen `errorCode` und`errorMessage`. Das folgende Beispiel zeigt einen abgelehnten Aufruf.

```
{
    "Records": [
        {
            "eventVersion": "1.11",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "ARO01EXAMPLEID:session-name",
                "arn": "arn:aws:sts::123456789012:assumed-role/app-invoke-role/session-
name",
                "accountId": "123456789012",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "ARO01EXAMPLEID",
                        "arn": "arn:aws:iam::123456789012:role/app-invoke-role",
                        "accountId": "123456789012",
                        "userName": "app-invoke-role"
```

```
                    },
                    "attributes": {
                        "creationDate": "2026-09-22T18:39:55Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2026-09-22T18:42:13Z",
            "eventSource": "lambda.amazonaws.com",
            "eventName": "InvokeWebFunctionEndpoint",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "203.0.113.42",
            "userAgent": "aws-sdk-python/1.34.0",
            "errorCode": "AccessDenied",
            "errorMessage": "User is not authorized to perform:
 lambda:InvokeWebFunctionEndpoint",
            "requestParameters": {
                "endpointArn": "arn:aws:lambda:us-east-1:123456789012:web-function/my-
fn/endpoint/my-endpoint"
            },
            "responseElements": null,
            "additionalEventData": {
                "functionRevisionArn": "arn:aws:lambda:us-east-1:123456789012:web-
function/my-fn/revision/rev-a1b2c"
            },
            "requestID": "c2e9b5d3-8f4e-4b2a-a07c-3d6e9f1b2345",
            "eventID": "e5d4c3b2-6a7f-8b9c-0d1e-2f3a4b5c6d7e",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "123456789012",
                    "type": "AWS::Lambda::WebFunction",
                    "ARN": "arn:aws:lambda:us-east-1:123456789012:web-function/my-fn"
                }
            ],
            "eventType": "AwsApiCall",
            "managementEvent": false,
            "recipientAccountId": "123456789012",
            "eventCategory": "Data",
            "tlsDetails": {
                "tlsVersion": "TLSv1.3",
                "cipherSuite": "TLS_AES_128_GCM_SHA256",
                "clientProvidedHostHeader": "bcdfghjklmnpqrst.b7t.lambda-web.us-
east-1.on.aws"
```

```
            }
        }
    ]
}
```

Verwaltungsereignisse in CloudTrail

CloudTrail zeichnet standardmäßig Verwaltungsereignisse für Webfunktionen auf. Sie können die Verwaltungsereignisse der letzten 90 Tage im CloudTrail Event-Verlauf einsehen, ohne eine Spur zu erstellen. Um sie länger zu speichern, erstellen Sie einen Trail- oder CloudTrail Lake-Event-Datenspeicher. Weitere Informationen finden Sie unter Protokollieren von Verwaltungsereignissen im AWS CloudTrail -Benutzerhandbuch.

CloudTrail zeichnet die folgenden Webfunktionsvorgänge als Verwaltungsereignisse auf. Das `eventName` Feld jedes Ereignisses entspricht dem Operationsnamen.

Ressource oder Aufgabe	Operationen
Web-Funktionen	`CreateWebFunction` , `GetWebFunction` , `ListWebFunctions` , `DeleteWebFunction`
Überarbeitungen	`CreateWebFunctionRevision` , `GetWebFunctionRevision` , `ListWebFunctionRevisions` , `DeleteWebFunctionRevision`
Endpunkte	`CreateWebFunctionEndpoint` , `GetWebFunctionEndpoint` , `ListWebFunctionEndpoints` , `UpdateWebFunctionEndpoint` , `DeleteWebFunctionEndpoint`
Kontoeinstellungen	`GetWebAccountSettings`
Ressourcenrichtlinien	`PutResourcePolicy` , `GetResourcePolicy` , `DeleteResourcePolicy`
Tags (Markierungen)	`TagResource` , `UntagResource` , `ListTags`

Der Aufruf einer Webfunktion ist eine Operation auf Datenebene. CloudTrail protokolliert ihn nur als Datenereignis, wie unter beschrieben. the section called “Datenereignisse der Webfunktion in CloudTrail”

Zugehörige Ressourcen

Weitere Informationen finden Sie in den folgenden Ressourcen:

- the section called “CloudTrail protokolliert”
- the section called “Berechtigungen”
- Erstellen Sie einen Trail für Ihren AWS-Konto

Beispielanwendungen für Webfunktionen

Jedes Beispiel verwendet `index.js` als Eingabedatei und überwacht den Host und den Port von`AWS_LAMBDA_HTTP_ENDPOINT`. Verpacken Sie die Anwendung mit ihren Produktionsabhängigkeiten und stellen Sie sie von einem versionierten Amazon S3 S3-Objekt aus bereit.

In der folgenden Tabelle sind die einzelnen Beispiele aufgeführt und was Sie daraus lernen können.

Beispiel	Benutze es, um zu lernen
the section called “Minimal Express Anwendung”	Starten Sie einen HTTP-Server mitExpress.
the section called “Ehre API und statische Dateien”	Stellen Sie API-Routen und statische Dateien mit bereitHono.
the section called “Server-sent Ereignisse”	Streamen Sie vom Server gesendete Ereignisse und beenden Sie die Arbeit, wenn der Client die Verbindung unterbricht.
the section called “Multi-Region Route zur Überprüfung”	Geben Sie bei der Validierung des Routing in mehreren Regionen die ausliefernde Region zurück.

Minimal Express Anwendung

Dieses Framework-basierte Beispiel ist eine Alternative zu dem Starter ohne Abhängigkeit, der von erstellt wurde. `--hello-world` Informationen zur Starter-Bereitstellung finden Sie unter. the section called “Stellen Sie Ihre Webfunktion bereit und testen Sie sie”

Installieren Sie Express:

```
npm init -y
npm install express
```

Erstellen Sie`index.js`:

```
const express = require('express');
const app = express();
const [host, port] = (process.env.AWS_LAMBDA_HTTP_ENDPOINT ||
 'localhost:3000').split(':');

app.get('/', (request, response) => {
  response.send('Hello from Lambda Web Functions!');
});

app.listen(port, host);
```

Hinweise zur Verpackung finden Sie unterthe section called “Ein Beispiel verpacken und bereitstellen”.

Ehre API und statische Dateien

Installieren Sie die Abhängigkeiten:

```
npm init -y
npm pkg set type=module
npm install hono @hono/node-server
```

Erstellen Sie`index.js`:

```
import { serve } from '@hono/node-server';
import { serveStatic } from '@hono/node-server/serve-static';
import { Hono } from 'hono';
import { secureHeaders } from 'hono/secure-headers';
```

```
const app = new Hono();
app.use(secureHeaders());
app.use('/static/*', serveStatic({ root: './' }));
app.get('/api/health', (context) => context.json({ status: 'ok' }));
app.notFound((context) => context.json({ error: 'not found' }, 404));

const [host, port] = (process.env.AWS_LAMBDA_HTTP_ENDPOINT ||
 'localhost:3000').split(':');

serve({ fetch: app.fetch, port: Number(port), hostname: host });
```

Platzieren Sie statische Dateien in einem `static` Verzeichnis und nehmen Sie das Verzeichnis in das Bereitstellungspaket auf.

Server-sent Ereignisse

Fügen Sie einer Express Anwendung die folgende Route hinzu:

```
app.get('/events', (request, response) => {
  response.setHeader('Content-Type', 'text/event-stream');
  response.setHeader('Cache-Control', 'no-cache');
  response.flushHeaders();

  const interval = setInterval(() => {
    response.write(`data: ${JSON.stringify({ time: Date.now() })}\n\n`);
  }, 1000);

  response.on('close', () => {
    clearInterval(interval);
  });
});
```

Die Anfrage bleibt aktiv, bis die Anwendung die Antwort beendet oder der Client die Verbindung unterbricht. Stellen Sie das Revisionstimeout für die maximale Stream-Dauer ein. WebSocketUpgrades werden nicht unterstützt.

Multi-Region Route zur Überprüfung

Fügen Sie eine Route hinzu, die die Region zurückgibt, in der die Anfrage ausgeführt wurde:

```
app.get('/region', (request, response) => {
```

```
  response.json({
    region: process.env.AWS_REGION,
    requestId: request.get('lambda-web-request-id')
  });
});
```

Rufen Sie die Route von den Client-Standorten aus auf, die für Ihre Anwendung wichtig sind. Für einen `MultiRegion` Endpunkt stammt die ausgewählte Region aus latenzbasiertem Routing und kann sich ändern. Verwenden Sie die Ausführungsumgebung nicht als Sitzungsspeicher.

Ein Beispiel verpacken und bereitstellen

```
npm ci --omit=dev
zip -r function.zip index.js package.json package-lock.json node_modules
aws s3 cp function.zip s3://amzn-s3-demo-bucket/samples/function.zip
```

Schließen Sie andere Dateien ein, die das ausgewählte Beispiel verwendet, z. B. das `static` Verzeichnis im Hono Beispiel. Informationen zum Erstellen einer Revision, die auf das hochgeladene Objekt verweist, finden Sie unter the section called “Kodieren Sie in Amazon S3” undthe section called “Überarbeitungen”. Informationen zum übergeordneten `deploy` Arbeitsablauf finden Sie unter. the section called “Eine Webfunktion erstellen (AWS CLI)” Sobald die Revision vorliegt `Active` und der Endpunkt sie bereitstellt, senden Sie eine Anfrage an die Endpunktdomäne, um zu bestätigen, dass das Beispiel reagiert. Informationen zur Testabdeckung, die über diesen Rauchtest hinausgeht, finden Sie unterthe section called “Testen”.

Bewährte Methoden für Lambda-Webfunktionen

Schützen Sie jeden Endpunkt

Jeder Endpunkt ist über das Internet erreichbar. Wie Sie also Anrufer authentifizieren, eine Eingangstür einrichten und welche Zugriffsrechte Sie haben, bestimmt, wer auf Ihre Anwendung zugreifen kann und worauf sie zugreifen kann. Verwenden Sie die folgenden Methoden:

- Jeder Endpunkt ist unabhängig von seinem Authentifizierungstyp über das Internet erreichbar. Mit `ApplicationManaged` authentifiziert Lambda keine Anfragen, sodass jeder Client, der die Domain kennt, Ihre Anwendung aufrufen kann. Implementieren Sie Authentifizierung, Autorisierung, Eingabevalidierung, Überprüfung der Anforderungsgröße und Schutz vor Missbrauch in der Anwendung oder an einer Vordertür.

- Verwenden Sie diese `IamAuth` Option, wenn Anrufer Anfragen signieren können und IAM-Richtlinien das erforderliche Autorisierungsmodell bereitstellen. Richten Sie die Ressourcenrichtlinien auf bestimmte Endpunkt-ARNs und Prinzipale ein.
- Stellen Sie Amazon CloudFront oder Amazon API Gateway vor einen Endpunkt, wenn Sie eine benutzerdefinierte Domain, AWS WAF Inspektion, Caching oder zentrale Edge-Kontrollen benötigen. Schränken Sie den Ursprung ein, damit Kunden die Eingangstür nicht umgehen können. Verwenden Sie mit CloudFront einen `IamAuth` Endpunkt und eine Zugriffskontrolle (Origin the section called “CloudFront Integration” Access Control, OAC), wie unter beschrieben. Bei anderen Eingangstüren ist der Endpunkt weiterhin direkt erreichbar. Überprüfen Sie daher, ob ein gemeinsam genutzter Schlüssel oder ein Token in der Anwendung verwendet wird.
- Verwenden Sie eine dedizierte Ausführungsrolle mit den geringsten Rechten. Trennen Sie die Bereitstellungsidentität, den Code-Uploader, den Lambda-Serviceprinzipal, die Ausführungsrolle und die Endpunkt-Aufruferberechtigungen voneinander.
- Speichern Sie keine Geheimnisse in Bereitstellungspaketen oder Protokollen. Laden Sie Geheimnisse zur Laufzeit aus einem geeigneten Geheimspeicher, speichern Sie sie nur so lange wie nötig im Cache und redigieren Sie sensible Anforderungsfelder.

Behalten Sie unveränderliche Einrichtungsartefakte bei

- Aktivieren Sie die Versionierung im Code-Bucket und notieren Sie die Versions-ID des Objekts für jede Bereitstellung.
- Überschreiben oder löschen Sie keine Objektversionen, die von aktiven Revisionen oder Rollback-Revisionen verwendet werden. Behalten Sie den Lambda-Serviceprinzipalzugriff so lange bei, wie eine Revision auf das Objekt verweist.
- Erstellen Sie Bereitstellungspakete in einer wiederholbaren Pipeline. Installieren Sie Produktionsabhängigkeiten aus einer Sperrdatei, schließen Sie Entwicklungsdateien aus und testen Sie das genaue Archiv vor dem Upload.
- Erstellen Sie native Abhängigkeiten für Arm64. Ein für x86_64 erstelltes systemeigenes Modul läuft nicht in der Ausführungsumgebung der Webfunktionen.
- Verwenden Sie separate Buckets, Präfixe, Rollen oder Konten, wenn Umgebungen unterschiedliche Vertrauens- oder Aufbewahrungsanforderungen haben.

Design für gleichzeitige Anfragen

- Behalten Sie anforderungsspezifische Daten im Umfang der Anfrage bei. Speichern Sie keine Anmeldeinformationen, Kundendaten, Antwortobjekte oder veränderbaren Status pro Anfrage in globalen Variablen.
- Verwenden Sie asynchron. I/O Synchrone CPU- oder Dateisystemarbeit blockiert die Node.js Ereignisschleife und verzögert jede laufende Anfrage in der Umgebung.
- Wird `worker_threads` für CPU-intensive Arbeiten verwendet, bei denen mehrere vCPUs oder weniger verwendet werden sollen, `maxConcurrencyPerEnvironment` um Konflikte zu vermeiden.
- Bewahren Sie den Node.js Heap getrennt vom Umgebungsspeicher auf. Die Umgebung hat 2 GB, während der Standard-V8-Heap 1.024 MB groß ist. Wenn Sie den Heap mit `--max-old-space-size` in start`NODE_OPTIONS`, lassen Sie Speicherplatz für Puffer, native Module, die Laufzeit und gleichzeitige Anfragen übrig.
- Verwenden Sie initialisierte Clients und Verbindungspools anforderungsübergreifend wieder, überprüfen Sie jedoch veraltete Verbindungen und gebundenes Cache-Wachstum. Behält den erforderlichen Status außerhalb der Ausführungsumgebung bei.
- Halten Sie ausgehende Verbindungen aufrecht, sodass Anfragen an Datenbanken und andere Abhängigkeiten sie wiederverwenden können, anstatt jedes Mal eine neue Verbindung zu öffnen. Node.jsaktiviert Keep-Alive standardmäßig in seinem globalen HTTP-Agenten. Wenn Sie Ihren eigenen Agenten erstellen oder einen SDK-Client konfigurieren, lassen Sie Keep-Alive aktiviert. Behandeln Sie auch Verbindungen, die von der Remote-Seite geschlossen werden, während sie inaktiv sind.
- Verwenden Sie eindeutige Namen für temporäre Dateien und entfernen Sie sie, wenn sie nicht mehr benötigt werden. Mehrere Anfragen teilen sich das 512 MB große `/tmp` Dateisystem der Umgebung, und Dateien können über mehrere Anfragen hinweg bestehen bleiben, bis die Umgebung entfernt wird.

Halten Sie den Start vorhersehbar

- Starten Sie den HTTP-Server umgehend und hören Sie sich den genauen Wert von an`AWS_LAMBDA_HTTP_ENDPOINT`.
- Halten Sie das Paket klein und entfernen Sie ungenutzte Produktionsabhängigkeiten.

- Initialisieren Sie nur die Ressourcen, die benötigt werden, bevor der Server Datenverkehr annehmen kann. Führen Sie optionale Arbeiten langsam durch, verhindern Sie jedoch, dass gleichzeitige Anfragen kostspielige Initialisierungen duplizieren.
- Übernehmen Sie `SIGTERM` die Bereinigung, die innerhalb von 1 Sekunde abgeschlossen ist, verlassen Sie sich jedoch nicht auf Shutdown-Callbacks, um die erforderlichen Daten dauerhaft zu speichern.

Legen Sie anhand des gemessenen Verhaltens Grenzwerte fest

- Belastungstest mit denselben Anwendungsabhängigkeiten und nachgelagerten Diensten, die in der Produktion verwendet werden. Erhöhen Sie die Parallelität, bis Latenz, CPU-Auslastung, Arbeitsspeicher oder Downstream-Kapazität zur Bindungsbeschränkung werden.
- Wird verwendet`maxConcurrencyPerEnvironment`, um Konflikte pro Prozess zu begrenzen, `maxEnvironments` um die horizontale Skalierung zu begrenzen und um nachgelagerte Abhängigkeiten vor Anforderungs-Bursts `rateLimit` zu schützen.
- Prüfen Sie vor Leistungstests und Starts die geltenden Konto- und Regionallimits. Durch die Endpunktkonfiguration kann keine Kapazität geschaffen werden, die über ein Kontolimit hinausgeht.
- Versuchen Sie es erneut mit HTTP 429-Antworten und HTTP 503-Antworten ohne `lambda-web-error-type` Header. Verwenden Sie Backoff und Jitter und wiederholen Sie den Vorgang nur, wenn der Vorgang idempotent ist. Verwenden Sie für Schreibvorgänge Idempotenzschlüssel oder Deduplizierung auf Anwendungsebene.

Bereitstellung mit expliziter Überprüfung und Rollback

- Verwenden Sie manuelle Revisionsgewichte für Produktionsendpunkte. Sorgen Sie für die automatische Bereitstellung der neuesten Version für Entwicklungsumgebungen, in denen ein sofortiger Rollout akzeptabel ist.
- Warten Sie, bis die neue Version verfügbar ist `Active` und das Endpunkt-Update abgeschlossen ist, bevor Sie es auswerten. Überprüfen Sie für `MultiRegion` `PerRegion` Endgeräte den Aktualisierungsstatus in jeder Region.
- Verwenden Sie bei Bereitstellungen auf Canary Anwendungsprotokolle oder benutzerdefinierte Messwerte, die die Revisions-ID enthalten. Service-vended Metriken enthalten keine Revisionsdimension.

- Bevor Sie den Traffic verlagern, definieren Sie die Metriken, die eine neue Version erfüllen muss, und definieren Sie, wie lange Sie sie einhalten (Bake-Kriterien), sowie die Schwellenwerte, die ein Rollback auslösen. Behalten Sie die Codeobjektversion, den Schlüsselzugriff und die Konfiguration der vorherigen Revision bei, bis das Rollback-Fenster endet. Lambda löscht automatisch die ältesten Revisionen, zu denen kein Endpunkt weiterleitet, wenn eine Webfunktion ihr Revisionslimit überschreitet. Behalten Sie die Rollback-Revision also auf einem Endpunkt (z. B. einem Testendpunkt) bei oder seien Sie bereit, sie aus derselben Objektversion neu zu erstellen. Details hierzu finden Sie unter the section called “Revisionsgrenzen und automatische Bereinigung”.

Erstellen Sie eine operative Ausgangsbasis

- Geben Sie strukturierte Anwendungsprotokolle mit Anforderungs-ID, Revisions-ID, Route, Statuscode, Dauer und den Identifikatoren aus, die für die Diagnose von Downstream-Ausfällen erforderlich sind. Protokollieren Sie keine Anmeldeinformationen, Token oder vertraulichen Anfragetexte.
- Alarmierung bei 5xx-Antworten und Drosselung sowie Überwachung der ausgeführten Anfragen, der aktiven Umgebungen und der Kontokapazität. Konfigurieren Sie wenige Fehlermetriken, um fehlende Daten angemessen zu behandeln.
- Verwenden Sie die Antwortanforderungs-ID und die Log-Stream-Header, um Client-Fehler mit Protokollen zu korrelieren. Aktivieren Sie DEBUG-Systemprotokolle nur, solange Sie die zusätzlichen Ereignisse benötigen.
- Erfassen Sie Verwaltungsvorgänge und Datenereignisse beim Aufrufen von Endpunkten AWS CloudTrail , wenn Ihre Prüfanforderungen sie benötigen.

Betrieb MultiRegion und PerRegion Endgeräte als verteilte Systeme

- Stellen Sie Anwendungsdaten, Geheimnisse, Schlüssel und Downstream-Abhängigkeiten in jeder Region zur Verfügung, die Datenverkehr empfangen kann.
- Testen Sie Fehler bei der teilweisen Bereitstellung und Fehler bei regionalen Abhängigkeiten. Gehen Sie nicht davon aus, dass in allen Regionen dieselbe Version ausgeführt wird, bis jedes regionale Update einen Erfolg meldet.
- Suchen Sie nach jedem Endpunkt-Update `updateStatus` nach jedem Eintrag in`regionalEndpoints`. Die Replikation kann in einigen Regionen fehlschlagen, während sie in anderen erfolgreich ist. Daher kann ein Update teilweise angewendet werden. Versuchen Sie

es erneut, oder führen Sie ein Rollback durch, bevor Sie die Bereitstellung als abgeschlossen betrachten.

- Testen Sie bei `PerRegion` Endgeräten Ihr externes DNS oder Ihr Front-Door-Failover unabhängig von der Bereitstellung der Web Function.
- Zentralisieren Sie die Betriebsansichten in der Heimatregion, behalten Sie jedoch einen Wiederherstellungspfad für Telemetrieverzögerungen oder -unterbrechungen bei.

Testen von Lambda-Webfunktionen

Web Functions laufen auf Standard-HTTP-Servern, sodass Sie das meiste Anwendungsverhalten lokal mit den normalen Tools Ihres Frameworks testen können. Cloud-Tests sind weiterhin erforderlich, um die IAM-Autorisierung, den Zugriff auf Amazon S3 und AWS Key Management Service (AWS KMS), den Lebenszyklusstatus von Webfunktionen, Skalierung, Telemetrie und das Verhalten in mehreren Regionen zu validieren.

Empfohlene Testschichten

In der folgenden Tabelle wird jede Testschicht beschrieben, was sie validiert und wo sie ausgeführt wird.

Testschicht	Was es validiert	Wo es läuft
Einheit	Geschäftslogik, Validierung, Autorisierungsentscheidungen, Serialisierung und Fehlerbehandlung.	Lokale Entwicklungs- und Build-Umgebungen.
HTTP-Integration	Routen, Middleware, Header, Streaming, Timeouts, reibungslose Verbindungsabbrüche und Framework-Verhalten.	Gegen einen lokal laufenden Produktionsserver.
Package	Eintragsdatei, Produktionsabhängigkeiten, Archivlayout, Start und Verwendung von durch den Server`AWS_LAMBDA_HTTP_ENDPOINT` .	Aus genau dem Archiv, das die Deployment-Pipeline hochlädt.
Cloud-Integration	IAM, Bucket-Richtlinien, Schlüssel, Lebenszyklusstatus, Endpunktverhalten, Telemetrie und AWS Downstream-Serviceanfragen.	In einer isolierten Umgebung AWS-Konto oder einer Testumgebung.

Testschicht	Was es validiert	Wo es läuft
Belastung und Belastbarkeit	Sicherheit bei Parallelität, Latenz, Drosselung, Downstream-Limits, Rollback und regionales Fehlerverhalten	In einer produktionsähnlichen Cloud-Umgebung mit explizite n Kosten- und Datenverk ehrskontrollen.

Führen Sie den HTTP-Server lokal aus

Verwenden Sie ein lokales Fallback für`AWS_LAMBDA_HTTP_ENDPOINT`, wie in der the section called “Minimal Express Anwendung” gezeigt. Sie können die Variable auch explizit setzen, um denselben Startpfad zu testen, der von Lambda verwendet wird:

```
AWS_LAMBDA_HTTP_ENDPOINT=127.0.0.1:3000 node index.js
```

Senden Sie Anfragen mit Ihrem normalen HTTP-Client und überprüfen Sie Statuscodes, Antworttexte, Header, Anforderungsvalidierung, Authentifizierungs-Middleware, Timeouts, Streaming und die Behandlung von Verbindungsabbrüchen beim Client.

Führen Sie gleichzeitige Anfragen während lokaler Integrationstests aus. Prüfen Sie den globalen Anforderungsstatus, gemeinsame veränderbare Caches, doppelte Initialisierungen, Dateinamenkollisionen und Blockierung von Ereignisschleifen.

Testen Sie das Bereitstellungspaket

1. Installieren Sie Abhängigkeiten aus der festgeschriebenen Sperrdatei und erstellen Sie das Archiv mit denselben Befehlen wie in der Bereitstellungspipeline.
2. Extrahieren Sie das Archiv in ein sauberes Verzeichnis. Führen Sie die Anwendung von diesem Verzeichnis aus aus, ohne Zugriff auf Entwicklungsabhängigkeiten oder Quelldateien ohne Nachverfolgung zu haben.
3. Stellen Sie sicher, dass die konfigurierte Eintragsdatei im erwarteten Pfad vorhanden ist, und startet den HTTP-Server.
4. Notieren Sie sich die Archivprüfsumme und die Amazon S3 S3-Objektversions-ID, die von der Revision verwendet wurden. Verwenden Sie diese Werte, um das genaue Artefakt zu identifizieren, das getestet wird.

Webfunktionen laufen auf Arm64. Wenn ein Paket native Abhängigkeiten enthält, erstellen und testen Sie diese Abhängigkeiten für Linux Arm64, anstatt Artefakte von einem x86-64-Entwicklungscomputer wiederzuverwenden. Sie können beispielsweise Produktionsabhängigkeiten in einem Arm64-Container installieren:

```
docker run --rm --platform linux/arm64 \
    --volume "$PWD":/app \
    --workdir /app \
    node:24 \
    npm ci --omit=dev
```

Testen Sie in einer isolierten Cloud-Umgebung

1. Laden Sie das getestete Archiv in einen versionierten Amazon S3 S3-Bucket hoch und erstellen Sie eine Revision, die auf die resultierende Objektversion verweist.
2. Fragen Sie die Revision ab, bis sie zu `Active` oder `Failed` wird. Bestehen Sie den Test mit den gemeldeten Daten nicht, `stateReason` anstatt auf unbestimmte Zeit zu warten.
3. Erstellen oder aktualisieren Sie einen Testendpunkt. Fragen Sie sowohl als `state` auch ab`updateStatus`. Überprüfen Sie für `MultiRegion` und `PerRegion` Endpunkte jeden regionalen Status.
4. Senden Sie Anfragen an den Endpunkt und überprüfen Sie die Antwort der Anwendung, die Protokolle`lambda-web-request-id`, die Metriken und die zu erwartenden Nebenwirkungen im nachgelagerten Bereich.
5. Löschen Sie den Endpunkt, die Funktion, die Testobjekte, Protokollgruppen und andere Testressourcen, wenn der Test abgeschlossen ist.

Autorisierung und Offenlegung von Tests

- Stellen Sie sicher`IamAuth`, dass unsignierte Anfragen fehlschlagen, zulässige Prinzipale erfolgreich sind, abgelehnte Prinzipale fehlschlagen und dass für den kontoübergreifenden Zugriff sowohl Identitäts- als auch Ressourcenrichtlinien erforderlich sind.
- Für`ApplicationManaged`, testen Sie fehlende, abgelaufene, falsch formatierte und unzureichende Anwendungsanmeldedaten. Testen Sie die Autorisierung getrennt von der Authentifizierung.
- Testen Sie Browser-Preflight-Anfragen, erlaubte und verweigerte Ursprünge, Anfragen mit Zugangsdaten und Sicherheitsheader, wenn Browser den Endpunkt aufrufen.

- Wenn ein Front-Door den Endpunkt schützt, vergewissern Sie sich, dass Anfragen mit direktem Ursprung verweigert werden und dass nur die erwartete Front-Door-Identität oder der erwartete Netzwerkpfad sie aufrufen kann.

Testen Sie Parallelität, Skalierung und Drosselung

- Erhöhen Sie die Last schrittweise und zeichnen Sie gleichzeitig Latenz, 4xx- und 5xx-Reaktionen, Drosselungen, aktive Umgebungen und Downstream-Auslastung auf.
- Testen Sie die konfigurierten und unabhängig `rateLimit` voneinander`maxEnvironments`. `maxConcurrencyPerEnvironment` Verwenden Sie den `lambda-web-error-type` Antwort-Header (`Platform.EndpointRequestThrottle`,`Platform.AccountRequestThrottle`, `Platform.EndpointMaxEnvThrottlePlatform.AccountActiveVCPUArm64Throttle`, oder`Platform.EndpointConcurrentInitThrottle`), um zu bestätigen, welches Limit jede HTTP 429-Antwort ausgelöst hat. Zeichnen Sie HTTP 503-Antworten, die keinen `lambda-web-error-type` Header haben, separat auf. Diese Antworten können auf einen vorübergehenden Lastverlust oder einen internen Fehler hinweisen. Details hierzu finden Sie unter the section called “Fehlertypen der Webfunktion”.
- Testen Sie Traffic Bursts und Kaltstarts, nicht nur den Durchsatz im stationären Zustand.
- Verwenden Sie eindeutige Testdaten und Idempotenzschlüssel. Versuchen Sie nicht, Anfragen, die nicht idempotent sind, während eines Auslastungstests automatisch erneut auszuführen.
- Legen Sie explizite Beschränkungen für Datenverkehr, Dauer und Kosten fest, bevor Sie einen Auslastungstest starten. Koordinieren Sie Tests, die sich auf Kontingente für gemeinsame Konten oder nachgelagerte Systeme auswirken können.

Testbereitstellungen und Rollback

Stellen Sie in einer Testumgebung zwei Versionen bereit und wenden Sie dasselbe gewichtete Routing-Verfahren an, das in der Produktion verwendet wird. Stellen Sie sicher, dass Ihre Protokolle oder benutzerdefinierten Messwerte Anfragen jeder Revision zuordnen können. Lösen Sie die Rollback-Bedingungen bewusst aus, stellen Sie den gesamten Datenverkehr auf die vorherige Version zurück und stellen Sie sicher, dass das Codeobjekt und die Abhängigkeiten weiterhin verfügbar sind.

Testen Sie für einen `MultiRegion` `PerRegion` OR-Endpunkt, ob ein teilweises Update fehlschlägt, und stellen Sie sicher, dass Ihre Bereitstellungsautomatisierung den ausgefallenen regionalen

Status erkennt. Testen Sie die Wiederherstellungsaktion und wiederholen Sie das Endpunkt-Update, nachdem Sie die Ursache behoben haben.

Fehlerbehebung bei Lambda-Webfunktionen

Beginnen Sie mit dem Ressourcenstatus, der HTTP-Antwort und den Protokollen für die Anfrage. Webfunktionen stellen einen separaten Status für die Funktion, die Revision, den Endpunkt und jede Region eines `MultiRegion` `PerRegion` OR-Endpunkts bereit. Ein Ausfall einer Ressource ändert nicht immer den Status der anderen.

> **Note**
>
> Bevor Sie die Diagnosebefehle in diesem Thema ausführen, stellen Sie sicher, dass Sie AWS CLI Zugriff und die Berechtigung zum Aufrufen der Lambda Web `get-*` Function-Operationen haben.

Sammeln von Diagnoseinformationen

1. Rufen Sie die Webfunktion, die Version und den Endpunkt ab. Notieren Sie jeden `statestateReason`,`updateStatus`, und `updateStatusReason` Wert.

   ```
   aws lambda-web get-web-function \
       --function-name my-web-app

   aws lambda-web get-web-function-revision \
       --function-name my-web-app \
       --revision-id revision-id

   aws lambda-web get-web-function-endpoint \
       --function-name my-web-app \
       --endpoint-name prod
   ```

   *my-web-app*Ersetzen Sie es durch Ihren Funktionsnamen, *revision-id* durch die zu prüfende Revision und *prod* durch Ihren Endpunktnamen.

2. Senden Sie eine Anfrage und notieren Sie den Statuscode und die Antwortheader.

   ```
   curl --include https://endpoint-domain/path
   ```

Der `lambda-web-request-id` Header identifiziert die Anfrage. Falls vorhanden, identifiziert der `lambda-web-logstream` Header den Amazon CloudWatch Logs-Protokollstream, der die Anfrage bearbeitet hat, und der `lambda-web-error-type` Header identifiziert einen Plattformfehler.

3. Überprüfen Sie die Anwendungs- und Systemprotokolle in der zentralen Protokollgruppe in der Heimatregion der Web-Funktion. Informationen zu Protokollgruppennamen und Anleitungen zur Korrelation finden Sie unterthe section called “Protokolle”.
4. Überprüfen Sie für `MultiRegion` und `PerRegion` Endpunkte jeden Eintrag in`regionalEndpoints`. Ein erfolgreicher Status auf Endpunktebene bedeutet nicht, dass das letzte Update in jeder Region erfolgreich war.

Probleme bei der Bereitstellung und beim Start

Symptom	Was muss überprüft werden	Auflösung
Eine Revision ist `Failed`	Die Revision`stateReas on` , die referenzierte Amazon S3 S3-Objekt version, die Bucket-Ri chtlinie, die Ausführun gsrolle, die Laufzeit und der AWS KMS Schlüssel.	Korrigieren Sie die zugrunde liegende Konfiguration und erstellen Sie eine neue Revision. Revisionen sind unveränderlich, und eine fehlgesch lagene Revision kehrt nicht zurück zu. `Active`
Der Endpunkt ist es`Active`, aber Anfragen schlagen beim Start der Anwendung fehl	System- und Anwendung sprotokolle für Eingabeda teifehler, Abhängigk eitsfehler oder einen Server, der nie zuhört. `AWS_LAMBDA_HTTP_EN DPOINT`	Führen Sie die verpackte Anwendung lokal aus, überprüfen Sie die Eingabeda tei und die Produktionsabhängigkeiten und hören Sie genau den Host und Port ab, den Lambda bereitstellt. Stellen Sie die Korrektur als neue Revision bereit.
Ein Endpunkt-Update schlägt fehl	`updateStatusReason` auf dem Endpunkt und, für `MultiRegion`	Korrigieren Sie den gemeldeten Abhängigkeits- oder Regionalfehler und rufen Sie dann `UpdateWeb`

Symptom	Was muss überprüft werden	Auflösung
	`PerRegion` Endpunkte, auf jedem regionalen Eintrag.	`FunctionEndpoint` erneut an, um das Update erneut zu versuchen.
`does not have s3:GetObject` auf `create-web-function-revision`	Der Code wird aus Amazon S3 gelesen, und entweder der Bucket-Richtlinie oder der Identitätsrichtlinie des Aufrufers fehlt der Lesezugriff.	Grant `s3:GetObject` und sowohl `s3:GetObjectVersion` in der Bucket-Richtlinie (für den Lambda-Service) als auch in der Identitätsrichtlinie des Aufrufers.
`AccessDenied` auf `get-web-function-endpoint`	Die IAM-Richtlinie `Resource` beinhaltet nicht den Endpunkt.	`web-function/` *`name`*`/*`In die Richtlinie `Resource` aufnehmen.

Um wiederholte Bereitstellungsfehler zu vermeiden, überprüfen Sie die Konfiguration und den Start in einer Testrevision, bevor Sie eine Produktionsrevision erstellen.

CLI- und Paketfehler

Fehler oder Symptom	Ursache	Auflösung
`invalid choice: lambda-web`	Die installierte Version ist AWS CLI älter als die Befehle der Webfunktion.	Aktualisieren Sie die AWS CLI und überprüfen Sie, ob die Web Function-Operationen aws lambda-web help aufgeführt sind.
`Unknown parameter: BuildConfig`	Die Anfrage verwendet PascalCase Mitglieds namen.	Verwenden Sie die in diesem Handbuch aufgeführten Mitglieder der CamelCase-Anfrage, z. B. `buildConfig codeConfig` , und. `serviceConfig`

Fehler oder Symptom	Ursache	Auflösung
`Cannot find module '/var/task/index.js'`	Die erforderliche Eintragsdatei fehlt im Archivstamm.	`index.js`Im Stammverzeichnis des Bereitstellungspakets platzieren. Schließt alle importierten Dateien und Produktionsabhängigkeiten ein.
`ERR_MODULE_NOT_FOUND` für eine Anwendungsdatei	Das Archiv lässt eine Quell- oder Build-Datei aus, die von `index.js` importiert wurde.	Untersuchen Sie das Archiv mit`unzip -l`, fügen Sie die fehlende Datei hinzu und erstellen Sie eine neue Version.
`Application.TimeoutListenError` , oder HTTP 502 bei der ersten Anfrage	Die Anwendung überwacht eine reine Loopback-Adresse, z. B. `127.0.0.1` oder`localhost` , oder einen anderen Port als den in. `AWS_LAMBDA_HTTP_ENDPOINT`	Analysiert `AWS_LAMBDA_HTTP_ENDPOINT` und überwacht genau diesen Host und Port.
`EMFILE` oder `too many open files`	Die Anwendung gibt bei gleichzeitigen Anfragen Sockets, Dateien oder Datenbankverbindungen preis.	Schließt anforderungsspezifische Handles, gebundene Verbindungspools und überwacht offene Handles bei anhaltender Parallelität.
`JavaScript heap out of memory`	Die Anwendung hat das standardmäßige V8-Heap-Limit erreicht, das unter dem Speicherlimit von 2 GB der Ausführungsumgebung liegt, oder gleichzeitige Anfragen haben den verbleibenden Umgebungsspeicher erschöpft.	Suchen Sie nach beibehaltenen Objekten und unbegrenzten Caches und reduzieren Sie die Parallelität. Wenn die Anwendung einen größeren Heap benötigt, legen Sie mit in ein größeres V8-Heap-Limit fest. `--max-old-space-size` `NODE_OPTIONS` Lassen Sie Speicherplatz für Puffer, systemeigene Zuweisungen und die Laufzeit übrig.

Fehler oder Symptom	Ursache	Auflösung
Abhängigk eitCannot find module//ERR_MODUL E_NOT_FOUND zur Laufzeit	node_modules war nicht im Bereitste llungspaket enthalten.	Installieren Sie Produktionsabhängi gkeiten und nehmen Sie sie in das Paket auf: führen Sie npm ci --omit=dev es aus und bestätigen Sie dann mitunzip -l function.zip .
Statische Dateien geben HTTP 404 zurück	Die Anwendung löst Dateipfade relativ zum Arbeitsverzeichnis auf, oder die Dateien wurden nicht gepackt.	Löst Pfade aus dem Modulverzeichnis (__dirname in CommonJS- oder import.meta.dirname ES-Module n) auf und vergewissere dich, dass sich die Dateien im Paket mit befinden. unzip -l
ESM-Importfehler, z. B. „Die Importanweisung kann nicht außerhalb eines Moduls verwendet werden“	package.j son deklariert den Modultyp nicht.	Füge CommonJS "type": "module" package.json hinzu oder verwende es konsistent.

Fehler bei HTTP-Anfragen

Antwort	Häufige Ursache	Auflösung
HTTP 400	Der Client hat eine nicht unterstützte HTTP-Methode verwendet oder eine Anfrage gesendet, die der Endpunkt nicht verarbeiten konnte.	Verwenden Sie eine unterstützte Methode und überprüfen Sie die Header und Grenzwerte der Anfrage in. the section called “HTTP-Verhalten”
HTTP 403	Die Autorisierung einer IamAuth Anfrage ist aus einem der folgenden Gründe fehlgeschlagen:	Vergewissern Sie sich, dass die Anfrage den erforderlichen X-Amz-Content-SHA2 56 Header enthält, überprüfen Sie die Signatur und überprüfen Sie lambda:In vokeWebFunctionEndpoint die

Antwort	Häufige Ursache	Auflösung
	• Die Anfrage ist nicht signiert. • In der Anfrage wird der erforderliche Inhalts-Hash-Header weggelassen. • Die Anfrage hat eine ungültige Signatur. • Der Anrufer hat keine Erlaubnis. Bei kontoüber greifenden Anrufen fehlt möglicherweise entweder die Identitätsrichtlinie oder die ressourcenbasierte Richtlinie.	erforderlichen Identitäts- und Ressource nrichtlinien. Informationen zu den Signatura nforderungen finden Sie unterthe section called “Signieren von Anforderungen”. Informationen zu den Genehmigungsanforderungen finden Sie unterthe section called “Einem Principal die Erlaubnis zum Aufrufen erteilen”.
HTTP 404	Die Anwendung definiert die angeforderte Route nicht, oder der Client verwendet eine veraltete Endpunktdomäne. Der `Host` Header stimmt möglicherweise auch nicht mit der Endpunktdomäne überein. Wenn Sie eine Fronttür wie Amazon so konfigurieren CloudFront , dass sie den `Host` Header des Viewers an den Ursprung weiterleitet, gibt Lambda diesen Fehler mit der `lambda-web-error-type` Einstellung auf `Platform.InvalidEndpoint` zurück.	Überprüfen Sie die aktuellen Routen`domainName` , den Anforderungspfad und die Anwendungsrouten. Verwenden Sie `lambda-web-error-type` und protokollieren Sie, um eine Plattformantwort von einer Anwendungsantwort zu unterscheiden.

Antwort	Häufige Ursache	Auflösung
HTTP 409	Der Endpunkt ist es nicht`Active`, oder der Endpunkt wurde zu einer Revision aufgelöst, die es nicht ist`Active`.	Untersuchen Sie die Revision und den Endpunktstatus. Wenn der Endpunkt der Fall ist`Pending`, warten Sie und versuchen Sie es erneut. Wenn ja`Failed`, erstellen Sie einen neuen Endpunkt. Entfernen Sie bei Revisionen fehlgeschlagene oder inaktive Revisionen aus den Endpunkt-Gewichtungen und erstellen Sie bei Bedarf eine Ersatzversion.
HTTP 429	Einer der folgenden Punkte ist aufgetreten: • Die Anfrage hat ein Endpunkt-Ratenlimit überschritten. • Der Endpunkt hat keine Umgebungskapazität mehr. • Das Konto hat ein regionales Anforderungs- oder vCPU-Limit erreicht. • Zu viele Ausführungsumgebungen wurden gleichzeitig für den Endpunkt initialisiert.	Lesen Sie den `lambda-web-error-type` Header, um herauszufinden, welches Limit angewendet wurde. Siehe the section called “Fehlertypen der Webfunktion”. Prüfen Sie die Drosselungskennzahlen (`EndpointRequestThrottles` `AccountRequestThrottles` ,`EndpointActiveEnvironmentThrottles` , und`AccountActiveVCPUArm64Throttles`), die Kennzahlen zur aktiven Kapazität, die Endpunkteinstellungen und die angewendeten Kontolimits. Durch die Drosselung (`Platform.EndpointConcurrentInitThrottle`) bei gleichzeitiger Initialisierung wird eine Drosselungsmetrik nicht erhöht. Versuchen Sie es mit Backoff und Jitter nur mit idempotenten Anfragen erneut.

Antwort	Häufige Ursache	Auflösung
HTTP 502	Ihre Anwendung ist ausgefallen, bevor sie mit der Bearbeitung von Anfragen begonnen hat, Lambda konnte die Ausführungsumgebung für Ihre Webfunktion nicht vorbereiten, oder Ihre Anwendung hat eine ungültige Antwort zurückgegeben.	`lambda-web-error-type` Verwendungs- und Systemprotokolle zur Identifizierung der Ursache. Siehe the section called “Fehlertypen der Webfunktion”.
HTTP 503	Die Anwendung ist bei der Bearbeitung einer Anfrage abgestürzt, der Endpunkt ist deaktiviert oder Lambda konnte keine Kapazität abrufen oder die Anwendung erreichen.	Verwenden Sie`lambda-web-error-type` , sofern vorhanden, Anwendungs- und Systemprotokolle, um die Ursache zu ermitteln. Siehe the section called “Fehlertypen der Webfunktion”. Ein 503 ohne `lambda-web-error-type` kann auf einen vorübergehenden Lastabwurf oder einen internen Fehler hinweisen. Versuchen Sie in beiden Fällen erneut, idempotente Anfragen mit Backoff und Jitter auszuführen.
HTTP 504	Die Anwendung hat innerhalb des Revisions-Timeouts (`Runtime.RequestTimeout`) nicht geantwortet.	Überprüfen Sie die Revision `timeoutSeconds` und die Downstream-Latenz.
Andere HTTP 5.xx	Die Anwendung hat den Status 5xx zurückgegeben, oder Lambda ist auf einen internen Fehler gestoßen (kein `lambda-web-error-type` Header).	Verwenden Sie die Anforderungs-ID, den Log-Stream-Header, Anwendungsprotokolle und Systemprotokolle, um die Quelle zu identifizieren und den Zustand des Prozesses zu überprüfen.

Antwort	Häufige Ursache	Auflösung
Bei einem großen Upload bleibt die Anfrage hängen oder es kommt zu einem Timeout. In den Protokollen werden 0 empfangene Byte angezeigt	Der Handler begann mit dem Schreiben der Antwort, während der Text der Anfrage noch eintraf. Der Anforderungspfad der Webfunktion ist Halbduplex: Lambda leitet den gesamten Anforderungstext weiter, bevor es die Antwort Ihrer Anwendung liefert.	Lesen Sie den Anfragetext vollständig, bevor Sie die Antwort schreiben. Textkörper mit einer Größe von weniger als 1 MB passen in Socket-Puffer und können dies maskieren.

Grenzwerte für Header anfordern

Für Anforderungsheader gelten zwei Grenzwerte. Lambda begrenzt den Header-Block auf 128 KiB und lehnt größere Anfragen ab, bevor sie Ihre Funktion erreichen. Der Node.js HTTP-Server in Ihrer Funktion begrenzt auch die Anzahl der Header-Zeilen, die er akzeptiert. Beide geben HTTP 431 Request Header Fields Too Large zurück, wie in der folgenden Tabelle dargestellt. Informationen zu den Plattformbeschränkungen finden Sie unterthe section called “Header-Grenzwerte”.

Symptom	Ursache	Auflösung
HTTP 431 mit`lambda-web-error-type: Platform.HeaderTooLarge` . Kein Aufruf und kein Eintrag im Anwendungsprotokoll.	Der Anforderungskopf überschreitet 131.072 Byte (128 KiB), gemessen am gesamten Anforderungskopf, wenn der Client ihn sendet.	Reduzieren Sie die Größe der Anforderungsheader. Dieses Limit kann nicht erhöht werden.
HTTP 431 mit `lambda-web-request-id` und `lambda-web-logstream` aber	Die Anfrage hat mehr als 1.000 Header-Zeilen, wie sie vom Node.js Server empfangen wurden, einschlie	Erhöhen `server.maxHeadersCount` Sie Ihren Servercode oder reduzieren

Symptom	Ursache	Auflösung
nein. `lambda-web-error-type` Der Handler läuft nicht.	ßlich der Header, die Lambda hinzufügt.	Sie die Anzahl der vom Client gesendeten Header-Zeilen.

Die Größenbeschränkung zählt jedes Byte des Anforderungskopfs, wenn der Client ihn sendet. Dazu gehören die Anforderungszeile, jede Header-Zeile (einschließlich `Host` aller Header, die die Client-Bibliothek hinzufügt) und die Zeilenabschlüsse. Header, die Lambda hinzufügt, werden nicht auf die Größenbeschränkung angerechnet.

Wenn eine Anforderung die Größenbeschränkung überschreitet, lehnt Lambda sie ab, bevor sie Ihre Funktion erreicht, sodass die Anforderung keinen Aufruf und keinen Eintrag im Anwendungsprotokoll erzeugt. Der `lambda-web-error-type` Antwort-Header identifiziert die Ursache.

Bei der Node.js Zählbegrenzung werden Kopfzeilen gezählt, keine eindeutigen Header-Namen. Ein Header, der dreimal wiederholt wird, zählt dreimal. Daher kann eine Anfrage, die Header sammelt `x-forwarded-for` oder über eine Kette von Proxys verfolgt, das Limit erreichen. Damit das passiert, muss kein einzelner Header ungewöhnlich aussehen. Das Limit wird anhand der Anfrage gemessen, die Ihre Funktion empfängt, einschließlich der von Lambda hinzugefügten Header. Das hat zur Folge, dass ein Client, der etwas weniger als 1.000 Header-Zeilen sendet, diesen Wert erreichen kann. In diesem Fall erreicht die Anfrage Ihre Ausführungsumgebung und der Node.js Server lehnt sie beim Parsen ab. Ihr Handler läuft nicht. Die Antwort enthält die `lambda-web-logstream` Header `lambda-web-request-id` und, aber sie enthält nicht den `lambda-web-error-type` Header.

Wenn Ihre Anwendung mehr Kopfzeilen akzeptieren muss, als die Node.js Standardeinstellung zulässt, erhöhen Sie das Node.js Limit in Ihrem Servercode. Stellen Sie `server.maxHeadersCount` die Anzahl der Kopfzeilen ein, die akzeptiert werden sollen, oder setzen Sie sie auf, `0` um die Node.js Zählprüfung zu deaktivieren. Die Größenbeschränkung von 128 KiB gilt weiterhin.

```
const server = http.createServer(handler);
server.maxHeadersCount = 2000;
server.listen(port, host);
```

Sie müssen die Größenbeschränkung für Node.js Header nicht ändern. In den verwalteten Node.js Laufzeiten ist es so eingestellt, dass es dem Plattformlimit entspricht, sodass das Plattformlimit für die Header-Größe gilt. Sie können das Plattformlimit nicht anhand des Anwendungscodes erhöhen.

Um die Grenzwerte einzuhalten, verschieben Sie den Massenstatus aus den Headern in den Anforderungstext, kürzen Sie Cookies und reduzieren Sie redundante Proxy- und Tracing-Header, bevor die Anfrage den Endpunkt erreicht. Wenn sich eine CloudFront Distribution vor dem Endpunkt befindet, leiten Sie nur die Header weiter, die Ihre Anwendung in der ursprünglichen Anforderungsrichtlinie benötigt.

Fehlertypen der Webfunktion

Wenn eine Anfrage an einen Webfunktions-Endpunkt fehlschlägt, gibt Lambda einen `lambda-web-error-type` Antwortheader zurück, der angibt, was schief gelaufen ist. Der Wert verwendet das Format`<Prefix>.<ErrorReason>`: Das Präfix gibt an, wo der Fehler aufgetreten ist, und der Grund gibt die Ursache an.

Präfix	Wo der Fehler aufgetreten ist
`Platform.`	Die Plattformschicht von Lambda, die Anfragen weiterleitet und validiert, Berechtigungen überprüft und Grenzwerte durchsetzt.
`Runtime.`	Die verwaltete Laufzeit, die Ihre Funktion lädt und ausführt.
`Application.`	Ihr Funktionscode.

`Runtime.RequestTimeout`Bedeutet beispielsweise, dass die verwaltete Laufzeit gemeldet hat, dass Ihre Funktion nicht innerhalb des konfigurierten Timeouts reagiert hat, und `Application.InitCodeError` dass Ihr eigener Initialisierungscode einen Fehler ausgelöst hat.

Lambda schließt den `lambda-web-error-type` Header ein, wenn der Fehler durch die Anfrage selbst oder durch Ihre Funktion, ihre Konfiguration oder ihren Code verursacht wurde. Lambda lässt den Header für interne Dienstfehler und für Autorisierungsfehler weg, sodass einem nicht autorisierten Anrufer keine Details preisgegeben werden. Ein Wert weist auf ein Problem auf dem Pfad zwischen der Lambda-Plattform und Ihrer Funktion hin. Im Anforderungspfad gibt der Fehler an, wo die Anfrage fehlgeschlagen ist. Verwenden Sie im Antwortpfad den Wert zusammen mit Ihrer Anwendungstelemetrie, um zu verstehen, was passiert ist.

Fehlertyp	Description
`Platform.MalformedRequest`	Lambda konnte die Anfrage nicht analysieren, oder die Anforderungszeile oder Methode wurde nicht akzeptiert.
`Platform.MissingHostHeader`	Die Anfrage enthielt keinen Host-Header, sodass Lambda den Zielendpunkt nicht auflösen konnte.
`Platform.InvalidEndpoint`	Der Host-Header wurde nicht zu einem gültigen, vorhandenen Endpunkt aufgelöst.
`Platform.HeaderTooLarge`	Der Header-Abschnitt der Anfrage hat die Größenbeschränkung für Lambda-Header überschritten.
`Platform.DisallowedHeader`	Die Anfrage enthielt einen Header, den Lambda nicht akzeptiert. Lambda gibt HTTP 400 zurück.
`Platform.UnsupportedHttpVersion`	Die Anfrage verwendete eine andere HTTP-Version als die HTTP/1.1, die Webfunktionen erfordern.
`Platform.CannotAssumeRole`	Lambda konnte die Ausführungsrolle Ihrer Funktion nicht übernehmen.
`Platform.KMSDecryptionError`	Lambda konnte den konfigurierten AWS KMS Schlüssel nicht verwenden, um Ihre Umgebungsvariablen oder andere Daten zu entschlüsseln, die für die Initialisierung Ihrer Funktion erforderlich sind.
`Platform.InvalidNetworkConfiguration`	Eine VPC-Netzwerkressource, von der Ihre Funktion abhängt, ist ungültig. Informationen zur Ursache finden Sie unterthe section called “Netzwerk- und Verbindungsprobleme”.
`Platform.RevisionNotActive`	Die Funktionsrevision, auf die die Anforderung abzielt, befindet sich nicht im Status Aktiv. Die Revision ist möglicherweise noch aktiviert oder konnte möglicherweise nicht aktiviert werden. Überprüfen Sie den Status der Revision. Wenn die Aktivierung noch läuft, versuchen Sie es nach einer kurzen Verzögerung erneut.

Fehlertyp	Description
`Platform.EndpointNotActive`	Der Endpunkt befindet sich nicht im Status Aktiv. Der Endpunkt wird möglicherweise noch bereitgestellt oder ist aus anderen Gründen nicht verfügbar. Überprüfen Sie den Status des Endpunkts. Wenn er noch erstellt wird, versuchen Sie es nach einer kurzen Verzögerung erneut.
`Platform.EndpointDisabled`	Der Wert des Endpunkts `rateLimit` ist 0, sodass Lambda alle Anfragen mit HTTP 503 ablehnt. Auf einen Wert ungleich Null setzen`rateLimit` , um den Verkehr wieder aufzunehmen.
`Platform.AccountDisabled`	Aufrufe von Webfunktionen sind für das Konto deaktiviert. Lambda gibt HTTP 403 zurück. Kontakt AWS Support.
`Platform.AccountRequestThrottle`	Das Kontingent für die Anforderungsrate auf Kontoebene wurde überschritten.
`Platform.EndpointRequestThrottle`	Das Kontingent für die Anforderungsrate des Endpunkts wurde überschritten.
`Platform.AccountActiveVCPUArm64Throttle`	Das aktive vCPU-Kontingent des Kontos wurde erreicht.
`Platform.EndpointMaxEnvThrottle`	Das maximale Umgebungslimit des Endpunkts wurde erreicht.
`Platform.EndpointConcurrentInitThrottle`	Zu viele Ausführungsumgebungen wurden gleichzeitig für den Endpunkt initialisiert. Versuchen Sie es erneut mit exponentiellem Backoff.
`Runtime.RequestTimeout`	Ihre Funktion hat innerhalb des konfigurierten Timeouts keine Antwort zurückgegeben.
`Runtime.OutOfMemory`	Ihre Funktion hat ihr Speicherlimit überschritten und der Prozess wurde gestoppt.

Fehlertyp	Description
`Runtime.Unreachable`	Der Runtime-Prozess ist abgestürzt, beendet oder die Verbindung verweigert.
`Runtime.Exit`	Der Laufzeitprozess wurde zwischen Anfragen beendet.
`Runtime.ExitError`	Der Laufzeitprozess wurde mit einem Fehler beendet.
`Runtime.InvalidResponse`	Die Antwortheader der Laufzeit haben den Lesepuffer überschritten.
`Runtime.OriginRequestError`	Der Anwendungsprozess ist bei der Bearbeitung einer Anfrage fehlgeschlagen. Einzelheiten finden Sie in den Anwendungsprotokollen.
`Runtime.ImportCodeError`	Die Runtime konnte Ihren Einstiegspunkt nicht importieren oder auflösen, z. B. ein fehlendes Modul oder ein fehlerhafter Pfad.
`Runtime.InitTimeout`	Ihre Funktion hat bei der Bereitstellung der Sandbox das Init-Timeout überschritten.
`Application.InitCodeError`	Ihr Initialisierungscode hat einen Fehler ausgelöst, bevor die Funktion mit der Bearbeitung von Anfragen begann.
`Application.TimeoutListenError`	Ihr Initialisierungscode hat nicht vor Ablauf der Initialisierungsfrist begonnen, den Port abzuhören.

Die Einschränkungsmetriken, die angeben, welcher Grenzwert angewendet wurde (für die `Platform.EndpointMaxEnvThrottle` FehlertypenPlatform.AccountRequestThrottle,Platform.EndpointRequestThrottle, und)Platform.AccountActiveVCPUArm64Throttle, finden Sie unter. the section called “Regionale Nutzungs- und Drosselungsmetriken”

Protokolle oder Metriken fehlen

- Stellen Sie sicher, dass die Ausführungsrolle Protokollgruppen und Streams erstellen und Protokollereignisse sowohl für die Zielprotokollgruppe als auch für die `.local` zugehörige Protokollgruppe speichern kann.

- Suchen Sie in der zentralen Protokollgruppe in der Heimatregion nach. Erstellen Sie keine Diagnosen, die von temporären Replikatregionen oder `.local` Protokollgruppen abhängen.
- Stellen Sie `systemLogLevel` es ein, `DEBUG` wenn Sie es benötigen. `platform.requestComplete`
- Lassen Sie nach dem Erstellen oder Aktualisieren von Ressourcen Verzögerungen bei der Telemetrieübertragung zu. Behandeln Sie das vorübergehende Fehlen eines Metrik- oder Protokollereignisses nicht als Beweis dafür, dass eine Anforderung nicht ausgeführt wurde.
- Denken Sie daran, dass die Metriken spärlich sind. Konfigurieren Sie gegebenenfalls Alarme so, dass fehlende Fehlermetriken als nicht fehlerhaft behandelt werden.

Wenn überhaupt keine Anwendungsprotokolle angezeigt werden, fehlen der Ausführungsrolle möglicherweise die Amazon CloudWatch Logs-Berechtigungen. Hängen Sie die `AWSLambdaBasicExecutionRole` verwaltete Richtlinie an die Ausführungsrolle an oder gewähren Sie `logs:CreateLogGrouplogs:CreateLogStream`, und`logs:PutLogEvents`. Die erforderlichen Berechtigungen finden Sie unter the section called “Rolle „Ausführung“”.

Probleme mit Codespeicherung und Verschlüsselung

Fehlende oder unzugängliche Amazon S3 S3-Objektversion

Vergewissern Sie sich, dass für den Bucket weiterhin die Versionierung aktiviert ist, dass die genaue Objektversion vorhanden ist und dass sowohl der Aufrufer als auch der Lambda-Serviceprinzipal über die erforderlichen Leseberechtigungen verfügen. Wenn eine aktive Revision den Zugriff auf ihre angeheftete Objektversion verliert, erstellen Sie eine neue Revision, die auf eine verfügbare Version verweist. Details hierzu finden Sie unter the section called “Kodieren Sie in Amazon S3”.

AWS KMS Der Zugriff ist fehlgeschlagen

Überprüfen Sie die Anruferberechtigungen, die Schlüsselrichtlinie, den Verschlüsselungskontext, den Schlüsselstatus und die Erlaubnis des Lambda-Serviceprinzipals. `kms:Decrypt` Stellen Sie für `MultiRegion` `PerRegion` Endgeräte sicher, dass ein Replikat des AWS KMS Schlüssels für mehrere Regionen vorhanden ist und in jeder Zielregion verwendet werden kann. Details hierzu finden Sie unter the section called “Verschlüsselung im Ruhezustand”.

Probleme mit und Endpunkten `MultiRegionPerRegion`

Verwenden Sie das `regionalEndpoints` Feld, um die betroffene Region zu identifizieren. Vergleichen Sie seine `stateReason` und `updateStatusReason` mit den gesunden Regionen. Prüfen Sie die regionalen Kontingente, die AWS KMS wichtigsten Replikate, die Laufzeitverfügbarkeit und den Replikationsstatus der Revision. Nachdem Sie die Ursache behoben haben, wiederholen Sie das Endpunkt-Update. Informationen zum Replikationsmodell finden Sie unterthe section called “Multi-Region Einsatz”.

Bei einem `MultiRegion` Endpunkt garantiert latenzbasiertes Routing (das Anfragen an die Region mit der niedrigsten gemessenen Latenz weiterleitet) nicht, dass jede Anfrage an die Region mit der niedrigsten Latenz gesendet wird. Testen Sie vom betroffenen Client-Standort aus und überprüfen Sie die Anwendungsprotokolle, um festzustellen, welche Region die Anfrage bearbeitet hat. Überprüfen Sie für einen `PerRegion` Endpunkt auch Ihre externe DNS- oder Front-Door-Routing-Konfiguration.

Netzwerk- und Verbindungsprobleme

Die meisten Verbindungsprobleme sind auf eine Serverversion zurückzuführen, die keine Connector-Referenz enthält oder auf eine ältere Connector-Version verweist, oder auf Connector-Subnetze, Sicherheitsgruppen oder Routen, die den Datenverkehr blockieren.

Verwenden Sie die folgende Reihenfolge, um ein Verbindungsproblem zu isolieren:

1. Führen Sie `get-web-function-endpoint`. Überprüfen Sie den automatischen Bereitstellungsmodus und überprüfen Sie, welche Version (en) Datenverkehr empfangen.
2. `get-web-function-revision`Für jede Serverversion ausführen. Vergewissern Sie sich, dass der Status lautet `Active` und dass er auf die erwartete Connector-Version verweist.
3. Führen Sie `aws lambda-core get-network-connector --identifier` mit dem Konnektor ARN aus der Revision aus`egressNetworkConnectorArn`. Vergewissern Sie sich, dass dies der Fall `State` ist `ACTIVE` und, falls der Connector aktualisiert wurde, dies `LastUpdateStatus` ist der Fall`Successful`. Verwenden Sie andernfalls `StateReasonCode` oder, `LastUpdateStatusReasonCode` um die Ursache zu ermitteln`InsufficientRolePermissions`, z. B.`InvalidSubnet`, oder`SubnetOutOfIPAddresses`.
4. Testen Sie das Ziel anhand der IP-Adresse, um die Erreichbarkeit des Netzwerks von der DNS-Auflösung zu trennen.
5. Überprüfen Sie die Connector-Subnetzrouten, Zielsicherheitsgruppen und Netzwerk-ACLs.

6. Überprüfen Sie die VPC-Flow-Logs in den Connector-Subnetzen.

Zu den häufigsten Symptomen gehören die folgenden.

Symptom	Check
Öffentliches Internet funktioniert, VPC-Ressourcen jedoch nicht	Die Serverversion hat keine Konnektorreferenz.
VPC-Zugriff funktioniert für einige Anfragen	Der Endpunkt leitet an zwei Revisionen mit unterschiedlichen Konnektorkonfigurationen weiter.
Ein Connector-Update hat keine Auswirkung	Die Serverversion verweist auf eine ältere Connector-Version.
Der öffentliche Zugriff wird nach dem Hinzufügen eines Connectors beendet	Die Connector-Subnetze verfügen nicht über die erforderliche NAT- oder IPv6-Route.
Timeout bei einem VPC-Ziel	Die Zielsicherheitsgruppe, die Routentabelle oder die Netzwerk-ACL blockiert die Verbindung.
Flow-Logs enthalten keine passenden Datensätze	Der Datenverkehr ist nicht in die VPC gelangt. Überprüfen Sie erneut die Serverversion und die Konnektorreferenz.
Private Namen schlagen fehl, aber direkte IP-Verbindungen funktionieren	Überprüfen Sie die VPC-DNS-Attribute und die Zuordnungen privater gehosteter Zonen.
IPv4 funktioniert, aber IPv6 schlägt fehl	Suchen Sie nach einer `DualStack` Connector-Version, IPv6-Routen und IPv6-Sicherheitsregeln.
Die Serverrevision ist`Active`, aber die Funktion kann keine privaten VPC-Ressourcen erreichen	Die Serverversion legt keinen Ausgangsnetzwerkanschluss fest ()`egressNetworkConnectorArn` . Lesen Sie das Feld mit`get-web-function-revision` ; wenn es leer ist, erstellen Sie eine Revision, die

Symptom	Check
	festlegtegressNetworkConnectorArn . Details hierzu finden Sie unter the section called “Netzwerk”.

Es gibt keinen interaktiven Zugriff auf eine Ausführungsumgebung für Webfunktionen. Verwenden Sie Anwendungsprotokolle, Endpunkt-Antwort-Header, Ressourcenstatusfelder, CloudWatch Metriken und VPC-Flow-Logs, um Fehler zu diagnostizieren.

Bereiten Sie ein Eskalationspaket vor

Wenn bei den vorherigen Prüfungen die Ursache nicht ermittelt wurde, sammeln Sie die folgenden Informationen, bevor Sie einen AWS Support Fall eröffnen. Auf diese Weise kann der Support-Techniker den Betrieb der Steuerungsebene, den regionalen Einsatz, die Anfrage und die Telemetrie miteinander korrelieren:

- Die AWS-Konto ID, die Heimatregion, der Funktionsname, die Revisions-ID, der Endpunktname, der Endpunkttyp und die Zielregionen.
- Der UTC-Zeitstempel und der Anforderungspfad für repräsentative Fehler, zusammen mit dem HTTP-Status `lambda-web-request-idlambda-web-logstream`, und den `lambda-web-error-type` Antwortheadern.
- Die Funktion, die Revision, der Endpunkt und die regionalen`state`, `stateReasonupdateStatus`, und `updateStatusReason` Werte.
- Relevante zentrale Anwendungs- und Systemprotokollereignisse, Drosselungs- und Kapazitätskennzahlen sowie die ungefähre Datenverkehrsrate zum Zeitpunkt des Auftretens des Problems.
- Die AWS CLI Version und eine redigierte Kopie der Verwaltungsanfrage. Entfernen Sie Anmeldeinformationen, Anwendungsgeheimnisse, Token und vertrauliche Anforderungsdaten.

Ähnliche Informationen

- the section called “Netzwerk”
- the section called “Überwachen”

AWS Lambda MicroVMs

Wenn Sie eine Anwendung erstellen, bei der mehrere Benutzer oder KI-Agenten eine Verbindung zu einer Computerumgebung herstellen und Code ausführen (z. B. interaktive Entwicklungsumgebungen, CI/CD Systeme oder Sandboxen für KI), benötigen Sie Rechenumgebungen, die schnell starten, zwischen den Mandanten isoliert bleiben und Sie keine Server oder Netzwerke verwalten müssen.

AWS Lambda MicroVMs wurden speziell für diese Anwendungsfälle entwickelt. Es handelt sich um serverlose Rechenumgebungen, die VM-level Isolierung mit vollen Betriebssystemfunktionen (z. B. Installation von Systempaketen oder Mounten von Dateisystemen), schnelle Startgeschwindigkeiten auf Snapshot-Basis und fein abgestimmter Steuerung der Eingangsnetzwerke (Portzugriff, Unterstützung für HTTP/2, gRPC) und Ausgangsnetzwerke (öffentlicher Internetzugang und VPC-Zugriff) bieten. WebSockets

Bei interaktiven Anwendungsfällen wie interaktiven Entwicklungsumgebungen und KI-Sandboxen kann es in Umgebungen zu langen Ruhephasen kommen, wenn der Kontext der Endbenutzer zu anderen Aufgaben wechselt oder wenn sie auf KI warten. MicroVMs können im Leerlauf angehalten werden, wodurch der Speicher- und Festplattenstatus erhalten bleibt und gleichzeitig die Kosten gesenkt werden. Wenn der Verkehr eintrifft, werden sie wieder aufgenommen. Sie können die automatische oder programmatische Unterbrechung und Wiederaufnahme aktivieren, um die Kosten im Leerlauf zu senken und gleichzeitig nahezu sofort einsatzbereit zu sein, wenn Ihre Endbenutzer zurückkehren.

AWS Lambda MicroVMs bieten diese Kernfunktionen über die Firecracker-Virtualisierung, die die Grundlage für die über 15 Billionen monatlichen Aufrufe von Lambda Functions bildet.

Um zu beginnen, sehen Sie sich Erstellen Sie Ihre erste MicroVM an. Preisinformationen finden Sie unter AWS Lambda – Preise. Informationen zu Sicherheit the section called “Sicherheit” und Berechtigungen finden Sie unter. Informationen zur Beobachtbarkeit finden Sie unterthe section called “Überwachen”. Hinweise zur Bedienung finden Sie unterthe section called “Bewährte Methoden”.

Wie funktionieren Lambda MicroVMS

1. Sie packen Ihren Anwendungscode und eine Datei `Dockerfile` in ein ZIP-Archiv und laden es auf Amazon S3 hoch.

2. Sie rufen die Lambda-API auf, um ein MicroVM-Image zu erstellen. Lambda führt Ihre aus`Dockerfile`, startet Ihre Anwendung und erfasst einen Snapshot der vollständig initialisierten Umgebung.
3. Wenn Ihre Anwendung eine isolierte Umgebung benötigt — für eine Benutzersitzung, einen Job oder eine Sandbox — rufen Sie an. `run-microvm` Lambda startet aus dem Snapshot heraus eine MicroVM mit schnellen Startzeiten.
4. Clients stellen über ihren dedizierten HTTPS-Endpunkt eine Verbindung zur laufenden MicroVM her. Es sind keine Load Balancer oder Ingress-Infrastruktur erforderlich.
5. Im Leerlauf wird die MicroVM angehalten, wodurch der Speicher- und Festplattenstatus erhalten bleibt und gleichzeitig die Kosten gesenkt werden. Wenn der Datenverkehr zurückkehrt, wird sie mit intaktem Speicher- und Festplattenstatus wieder aufgenommen. Suspend-resume Das Verhalten kann automatisch über Lebenszyklusrichtlinien konfiguriert oder direkt durch API-Aufrufe von und ausgelöst werden. `suspend-microvm` `resume-microvm`
6. Wenn die Sitzung endet, beenden Sie die MicroVM, um alle Ressourcen freizugeben.

Wann sollte Lambda MicroVMS verwendet werden

Lambda MicroVMS eignet sich gut für Anwendungsfälle, in denen Benutzer oder AI-generated Code ausgeführt werden und Ausführungsumgebungen erforderlich sind, die eine starke Isolierung, schnelle Latenz bei Start und Wiederaufnahme sowie Entwicklerkontrolle über den Lebenszyklus der Umgebung und die Beibehaltung des Zustands bieten. Zu den typischen Anwendungsfällen gehören:

- Interaktive Codeumgebungen — Entwicklungsumgebungen, in denen Benutzer Code in Echtzeit schreiben und ausführen.
- Sandboxen zur Ausführung von KI-Code — Kurzlebige Sandboxen für die sichere Ausführung von Code. AI-generated
- Datenanalyseanwendungen — Jupyter-Notebooks und kurzlebige Datenverarbeitungs-Workloads, die vom Benutzer bereitgestellte Skripts ausführen.
- Sicherheitsscans — Tools zur Schwachstellenanalyse, die isolierte Ausführungsumgebungen benötigen.
- Umgebungen für verstärktes Lernen — Isolierte Sandboxen für die Bewertung und Schulung von KI-Agenten, in denen bei jedem Durchlauf neue Umgebungen eingerichtet werden.
- Multi-tenant CI/CD— Task-Executoren, die eine Isolierung zwischen den Mandanten erfordern.

- Spieleserver — Hosting-Umgebungen, in denen vom Benutzer bereitgestellte Skripts mit starker Isolierung ausgeführt werden.

Schlüssel-Features

- Schneller Start — MicroVMs werden anhand vorinitialisierter Snapshots wieder aufgenommen, wobei die Anwendungsinitialisierung übersprungen wird. Siehe MicroVM-Images.
- Lebenszykluskontrolle — Mithilfe konfigurierbarer Richtlinien für den Leerlauf können Sie MicroVMs programmgesteuert oder automatisch aussetzen, fortsetzen und beenden. Siehe MicroVMS ausführen.
- Flexibles Netzwerk — MicroVMs unterstützen eingehenden HTTPS-Verkehr auf konfigurierbaren Ports und verfügen über eine vom Service bereitgestellte JWE-Authentifizierung. Der ausgehende Zugriff ist konfigurierbar. Greifen Sie auf das öffentliche Internet oder Ihre VPC zu. Einzelheiten finden Sie im Kapitel überNetzwerk.
- Flexible Ressourcenzuweisung — Stellen Sie jede MicroVM für die grundlegende oder durchschnittliche Nutzung bereit und bieten Sie die Flexibilität, bei Spitzenaktivitäten vertikal auf das Vierfache der konfigurierten Ausgangswerte zu skalieren. Zahlen Sie den Basistarif, während Ihre MicroVM läuft, und zahlen Sie nur für die aktive Nutzung, die über dem Basispreis liegt.

AWS Lambda MicroVMS-Kernkonzepte

AWS Lambda MicroVMS verwendet mehrere Ressourcentypen, die Sie erstellen und verwalten. Auf dieser Seite wird jeder Ressourcentyp beschrieben, wie Lambda Ihr MicroVM-Image zu einem Snapshot zusammenstellt, und der Lebenszyklus gibt an, den eine MicroVM zur Laufzeit durchläuft — die Grundlage für die Erstellung von Anwendungen mit MicroVMs.

Die wichtigsten Konzepte

MicroVM

Eine MicroVM ist eine Ressource, die eine isolierte Rechenumgebung für einen einzelnen Mandanten, eine Benutzersitzung oder einen Job darstellt. Auf jeder MicroVM wird ein Amazon Linux 2023-Betriebssystem mit Betriebssystemfunktionen ausgeführt, das einen nahezu sofortigen Start und Wiederaufnahme ermöglicht. MicroVMs empfangen Anfragen über eingehende HTTPS-Verbindungen und können im Leerlauf angehalten werden, wodurch der Speicher- und

Festplattenstatus erhalten bleibt. Eine angehaltene MicroVM wird wieder aufgenommen, wenn der Datenverkehr zurückkehrt.

MicroVM-Image

Ein MicroVM-Image ist eine Ressource, die die Anwendungsumgebung einer MicroVM definiert. Wenn Sie ein Image erstellen, baut Lambda es in einen Snapshot ein, der einen nahezu sofortigen Start ermöglicht (siehe unten). Wie Lambda Ihr Image erstellt

Um ein MicroVM-Image zu erstellen, stellen Sie ein Zip-Paket mit den Artefakten Ihrer Anwendung bereit, das auf Amazon S3 hochgeladen wurde. `Dockerfile` Sie müssen ein Lambda-published verwaltetes Basis-Image als Grundlage verwenden — geben Sie es mit dem `base-image-arn` Parameter an. Your `Dockerfile` definiert die Anwendungsebenen, die Lambda auf der verwalteten Basis erstellt.

MicroVM-Images sind versioniert. Jede Version stellt einen einzelnen Build dar, der aus einem bestimmten Code-Artefakt und einem Basis-Image erstellt wurde. Eine Version durchläuft den Build-Status (`PENDING`→ `IN_PROGRESS` → `SUCCESSFUL` oder`FAILED`), und erfolgreiche Versionen können auf `ACTIVE` oder gesetzt werden. `INACTIVE` Einzelheiten zu Image-Status und Verwaltung finden Sie unterMicroVM-Images.

Netzwerkanschlüsse

Netzwerkkonnektoren sind Ressourcen, die steuern, wie der Datenverkehr Ihre MicroVM erreicht und wie Ihre MicroVM externe Dienste erreicht. Sie verknüpfen Connectors zur Laufzeit mit einer MicroVM, um den eingehenden und ausgehenden Zugriff unabhängig voneinander zu konfigurieren.

Build-time und Laufzeit-Connectors können unterschiedlich sein, sodass Ihre MicroVM während der Image-Erstellung und während der Laufzeit unterschiedliche Umgebungen erreichen kann.

Verwenden Sie Lambda-provided Standardwerte für den Zugriff auf eingehende Ports (mit JWE-Authentifizierung), den Shell-Zugriff und den öffentlichen Internetausgang. Erstellen Sie Ihren eigenen Netzwerkkonnektor, um ausgehenden Datenverkehr durch Ihre VPC zu leiten.

Wie Lambda Ihr Image erstellt

Wenn Sie ein MicroVM-Image erstellen oder aktualisieren, führt Lambda einen Build-Prozess durch, der einen Firecracker-Snapshot erstellt. Dieser Snapshot erfasst den vollständig initialisierten Zustand Ihrer Anwendung und ermöglicht den nahezu sofortigen Start und die Wiederaufnahme von MicroVMs, die von dort aus ausgeführt werden.

Der Build-Prozess:

1. Lambda stellt mithilfe des von Ihnen angegebenen verwalteten Basis-Image eine neue MicroVM bereit.
2. Lambda führt Ihre `Dockerfile` Anweisungen zur Installation von Abhängigkeiten und zur Konfiguration Ihrer Umgebung aus.
3. Lambda startet Ihre Anwendung mit dem `ENTRYPOINT` Befehl oder. `CMD`
4. Wenn Sie den `/ready` Hook aktiviert haben, wartet Lambda darauf, dass Ihre Anwendung die Bereitschaft signalisiert (HTTP 200).
5. Lambda erfasst einen Snapshot des Festplatten- und Speicherstatus, einschließlich aller laufenden Prozesse.

Wenn Sie eine MicroVM ausführen, stellt Lambda sie aus diesem Snapshot wieder her. Ihre Anwendung wird aus dem vorinitialisierten Zustand wieder aufgenommen, ohne den Start zu wiederholen.

Wenn Ihre Anwendung während des Builds eindeutige Inhalte generiert (z. B. eindeutige IDs, Geheimnisse oder Netzwerkverbindungen), wird dieser Inhalt von allen MicroVMs gemeinsam genutzt, die auf derselben Image-Version ausgeführt werden. Um dies zu vermeiden, generieren Sie eindeutigen Inhalt, nachdem die MicroVM begonnen hat, den Lifecycle-Hook zu verwenden. `/run` Einzelheiten finden Sie im Abschnitt zur Snapshot-Kompatibilität unterMicroVM-Images.

MicroVM-Lebenszyklus

Zur Laufzeit durchläuft eine MicroVM die folgenden Phasen:

1. Lauf — Du rufst an. `run-microvm` Lambda stellt die MicroVM aus dem Image-Snapshot wieder her, weist eine eindeutige ID zu und erstellt einen Endpunkt. Die MicroVM wechselt von zu. `PENDING` `RUNNING`
2. Wird ausgeführt — Ihre Anwendung empfängt und verarbeitet Anfragen über ihre Endpunkt-URL.
3. Aussetzen — Nach einer konfigurierbaren Leerlaufzeit (oder über die `suspend-microvm` API) wechselt die MicroVM `SUSPENDING` zu`SUSPENDED`. Speicher und Festplattenstatus bleiben erhalten.
4. Fortfahren — Die MicroVM wechselt von `SUSPENDED` direkt zurück zu dem `RUNNING` Zeitpunkt, an dem der Verkehr eintrifft (falls`autoResumeEnabled=true`) oder Sie anrufen`resume-microvm`.

5. Beenden — Die MicroVM wechselt `TERMINATING` zu dem `TERMINATED` Zeitpunkt, an dem Sie anrufen `terminate-microvm` oder die maximale Dauer überschritten wird.

Zustände

In der folgenden Tabelle werden die einzelnen MicroVM-Zustände beschrieben. Diese Status ermöglichen es Ihnen, zuverlässige Anwendungen zu erstellen und eine angemessene Fehlerbehandlung zu implementieren.

Status	Description
`PENDING`	MicroVM wird bereitgestellt. Ressourcen werden zugewiesen und der Snapshot wird geladen.
`RUNNING`	MicroVM ist aktiv und akzeptiert Datenverkehr über seine Endpunkt-URL. Der `/run` Hook ist abgeschlossen.
`SUSPENDING`	MicroVM wird angehalten. Der `/suspend` Hook wird ausgeführt. Festplatte und Speicher werden überprüft.
`SUSPENDED`	MicroVM ist angehalten. Der Zustand bleibt erhalten. Es fallen keine Rechengebühren an. Kann wieder aufgenommen oder beendet werden.
`TERMINATING`	MicroVM wird beendet. Der `/terminate` Hook wird ausgeführt. Ressourcen werden veröffentlicht.
`TERMINATED`	MicroVM wurde beendet. Diese ist ein Terminalstatus. Die MicroVM kann nicht wieder aufgenommen oder neu gestartet werden.

Zustandsübergänge

Die folgende Tabelle zeigt die gültigen Übergänge zwischen MicroVM-Zuständen und was die einzelnen Übergänge auslöst.

Ursprungszustand	Zielzustand	Auslöser
`PENDING`	`RUNNING`	Die Bereitstellung ist abgeschlossen, der `/run` Hook war erfolgreich.

Ursprungszustand	Zielzustand	Auslöser
RUNNING	SUSPENDING	Die Leerlaufdauer wurde überschritten oder es wurde ein expliziter suspend-microvm API-Aufruf ausgeführt.
SUSPENDING	SUSPENDED	/suspendHook abgeschlossen, Speicher- und Festplattenstatus überprüft.
SUSPENDED	RUNNING	Der Datenverkehr kommt an (autoResumeEnabled=true) oder ein expliziter resume-microvm API-Aufruf.
RUNNING	TERMINATING	Expliziter terminate-microvm API-Aufruf oder maximumDurationInSeconds überschritten.
SUSPENDED	TERMINATING	suspendedDurationSeconds überschritten oder expliziter terminate-microvm API-Aufruf.
TERMINATING	TERMINATED	/terminate Hook abgeschlossen, alle Ressourcen freigegeben.

Important

Wenn Ihr /run Hook ausfällt oder ein Timeout auftritt, wechselt die MicroVM möglicherweise direkt zu dem Hook, TERMINATING ohne ihn jemals zu erreichenRUNNING. Implementieren Sie Timeout und Fehlerbehandlung in Ihren Hooks, um stille Ausfälle zu vermeiden.

Erstellen Sie Ihre erste Lambda MicroVM

Dieses Tutorial führt Sie durch die Erstellung eines AWS Lambda MicroVM-Images und das Ausführen einer MicroVM von diesem aus. Am Ende haben Sie eine laufende Anwendung, auf die über HTTPS zugegriffen werden kann.

Voraussetzungen

Sie benötigen zwei Dinge, bevor Sie beginnen:

1. Ein Amazon S3-Bucket in Ihrer bevorzugten AWS Region zum Speichern Ihres Anwendungsartefakts (die ZIP-Datei, die Sie in Schritt 2 erstellen).
2. Eine IAM-Build-Rolle, die Lambda bei der Image-Erstellung übernimmt. Lambda verwendet diese Rolle, um Ihr Code-Artefakt von Amazon S3 herunterzuladen und Build-Protokolle in Amazon S3 zu schreiben. CloudWatch

Erstellen Sie die IAM-Rolle mit der folgenden Vertrauensrichtlinie. Dadurch kann der Lambda-Service die Rolle übernehmen:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": { "Service": "lambda.amazonaws.com" },
    "Action": ["sts:AssumeRole", "sts:TagSession"]
  }]
}
```

Hängen Sie die folgende Berechtigungsrichtlinie an die Rolle an. `<your-bucket-name>`Ersetzen Sie durch Ihren Amazon S3-Bucket-Namen:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject"],
      "Resource": "arn:aws:s3:::<your-bucket-name>/*"
    },
    {
      "Effect": "Allow",
      "Action": ["logs:CreateLogGroup", "logs:CreateLogStream", "logs:PutLogEvents"],
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

> **Note**
>
> Wenn Ihre Daten `Dockerfile` aus einem privaten AWS ECR-Repository abgerufen werden, fügen Sie der Berechtigungsrichtlinie auch `ecr:GetAuthorizationToken` und `ecr:BatchGetImage` hinzu.

Erstellen Ihres ersten MicroVM-Images

Ein MicroVM-Image erfasst Ihre Anwendung in einem vollständig initialisierten Zustand. Wenn Sie eine MicroVM von diesem Image aus ausführen, wird Ihre Anwendung sofort gestartet — ohne Verzögerung beim Start oder bei der Initialisierung.

Schritt 1: Schreiben Sie Ihre Anwendung und Dockerfile

Erstellen Sie einen einfachen HTTP-Server, der in Ihrer MicroVM ausgeführt wird. In diesem Beispiel wird Folgendes Node.js ohne externe Abhängigkeiten verwendet:

`app.js`

```
// Minimal HTTP server — listens on port 8080
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({ status: 'ok', path: req.url }));
});

server.listen(8080, () => {
  console.log('Listening on port 8080');
});
```

Erstellen Sie als Nächstes `Dockerfile` ein Paket und starten Sie Ihre Anwendung:

`Dockerfile`

```
# Use a lightweight Node.js runtime for your application layers
FROM node:24-alpine

# Set the working directory inside the container
WORKDIR /app
```

```
# Copy your application code
COPY app.js .

# Declare the port your app listens on
EXPOSE 8080

# Start the application — Lambda snapshots the running state
CMD ["node", "app.js"]
```

> **Note**
>
> Die `FROM` Anweisung legt das Container-Image für Ihre Anwendungsebenen fest. Sie können jedes kompatible Container-Image verwenden. Das von Lambda verwaltete Basisimage (das das MicroVM-Betriebssystem und die Servicekomponenten bereitstellt) wird `--base-image-arn` in Schritt 3 separat angegeben.
> Wenn Ihre Anwendung eindeutige Werte (IDs, Geheimnisse oder kryptografisches Material) generiert, verwenden Sie die Standardbibliothek für den kryptografisch sicheren Pseudozufallszahlengenerator (CSPRNG) Ihrer Sprache, um die Eindeutigkeit aller MicroVMs sicherzustellen. Wenn Ihre Anwendung OpenSSL verwendet, verwenden Sie das Lambda-Basiscontainer-Image, das eine Snapshot-kompatible Version enthält. Einzelheiten finden Sie im Abschnitt zur Snapshot-Kompatibilität unter. MicroVM-Images

Schritt 2: Verpacken und auf Amazon S3 hochladen

Verpacken Sie Ihr `Dockerfile` Land in ein ZIP-Archiv `app.js` und laden Sie es dann in Ihren Amazon S3-Bucket hoch. Führen Sie die folgenden Befehle in Ihrem Terminal aus:

```
zip app.zip app.js Dockerfile
aws s3 cp app.zip s3://your-bucket-name/app.zip
```

Schritt 3: Erstellen Sie das MicroVM-Image

Rufen Sie `create-microvm-image` auf, um den Build zu starten. Lambda lädt Ihre Zip-Datei von Amazon S3 herunter, führt Ihre aus`Dockerfile`, startet Ihre Anwendung und erfasst einen Firecracker-Snapshot des vollständig initialisierten Zustands:

```
aws lambda-microvms create-microvm-image \
  --name my-first-microvm-image \
```

```
  --code-artifact uri=s3://your-bucket-name/app.zip \
  --base-image-arn arn:aws:lambda:us-east-1:aws:microvm-image:al2023-1 \
  --build-role-arn arn:aws:iam::123456789012:role/MicrovmBuildRole
```

Das Bild beginnt im Status. `CREATING` Überprüfen Sie den Build-Status mit:

```
aws lambda-microvms get-microvm-image \
  --image-identifier my-first-microvm-image
```

Wenn der Build abgeschlossen ist, ändert sich das `state` Feld in`CREATED`:

```
{
  "imageName": "my-first-microvm-image",
  "imageArn": "arn:aws:lambda:us-east-1:123456789012:microvm-image:my-first-microvm-
image",
  "state": "CREATED",
  "imageVersion": "1.0",
  ...
}
```

Wenn der Status lautet`CREATE_FAILED`, überprüfen Sie die Build-Protokolle CloudWatch unter`/aws/lambda-microvms/my-first-microvm-image`.

Führen Sie Ihre erste MicroVM aus

Sobald Ihr MicroVM-Image den `CREATED` Status erreicht hat, können Sie MicroVMs von dort aus ausführen. Jedes Image kann viele MicroVMs starten — eine pro Mandant, Benutzersitzung oder Job.

Führen Sie eine MicroVM mit dem folgenden Befehl aus:

```
aws lambda-microvms run-microvm \
  --image-identifier my-first-microvm-image \
  --ingress-network-connectors "arn:aws:lambda:us-east-1:aws:network-connector:aws-
network-connector:ALL_INGRESS" \
  --egress-network-connectors "arn:aws:lambda:us-east-1:aws:network-connector:aws-
network-connector:INTERNET_EGRESS" \
  --idle-policy
 '{"autoResumeEnabled":true,"maxIdleDurationSeconds":900,"suspendedDurationSeconds":300}'
```

Parameter erklärt:

- `--ingress-network-connectors`— Aktiviert eingehenden HTTPS-Verkehr zu Ihrer MicroVM auf allen Ports. Dies ist ein Connector Lambda-managed.
- `--egress-network-connectors`— Ermöglicht den ausgehenden Internetzugang von Ihrer MicroVM aus. Dies ist ein Connector Lambda-managed.
- `--idle-policy`— Konfiguriert das automatische Verhalten bei Unterbrechung und Wiederaufnahme. Diese Richtlinie setzt die MicroVM nach 15 Minuten Inaktivität aus, hält sie für bis zu 5 Minuten in den Ruhezustand und setzt sie automatisch fort, wenn Datenverkehr eintrifft.

Die Antwort enthält die MicroVM-ID und die Endpunkt-URL:

```
{
  "microvmId": "mvm-01234567-abcd-ef01-2345-6789abcdef01",
  "state": "PENDING",
  "endpoint": "mvm-01234567-abcd-ef01-2345-6789abcdef01.lambda-microvm.us-
east-1.on.aws",
  ...
}
```

Warten Sie, bis der Status Folgendes erreicht `RUNNING` hat:

```
aws lambda-microvms get-microvm \
  --microvm-identifier mvm-01234567-abcd-ef01-2345-6789abcdef01
```

Stellen Sie eine Verbindung zu Ihrer MicroVM her

Für alle Anfragen an einen MicroVM-Endpunkt ist ein Authentifizierungstoken erforderlich. Generieren Sie eine mit:

```
aws lambda-microvms create-microvm-auth-token \
  --microvm-identifier mvm-01234567-abcd-ef01-2345-6789abcdef01 \
  --expiration-in-minutes 30 \
  --allowed-ports '[{"allPorts":{}}]'
```

Die Antwort enthält ein Token im `authToken` Feld. Verwenden Sie es, um eine Anfrage an Ihre laufende Anwendung zu senden:

```
curl https://mvm-01234567-abcd-ef01-2345-6789abcdef01.lambda-microvm.us-east-1.on.aws/
 \
```

```
  -H "X-aws-proxy-auth: <token-value>"
```

Sie sollten die Antwort Ihrer Anwendung sehen:

```
{"status":"ok","path":"/"}
```

Ihre MicroVM läuft und verarbeitet den Datenverkehr. Die Anwendung, die Sie in Schritt 1 geschrieben haben, ist unter der Endpunkt-URL live.

Bereinigen

Um laufende Gebühren zu vermeiden, beenden Sie die MicroVM, wenn Sie fertig sind:

```
aws lambda-microvms terminate-microvm \
  --microvm-identifier mvm-01234567-abcd-ef01-2345-6789abcdef01
```

Nächste Schritte

- Erfahren Sie mehr über Schlüsselkonzepte den Snapshot-Prozess, die Lebenszyklusstatus und die Basis-Images.
- Erfahren MicroVM-Images Sie mehr über Image-Build-Hooks, Versionierung und Snapshot-Kompatibilität.
- SDK-Beispiele, Lifecycle-Hooks und Skalierungsstrategien finden MicroVMS ausführen Sie unter.
- Informationen zu the section called “Integrationen” unterstützten Serviceintegrationen finden Sie unter.

MicroVM-Bilder

In diesem Abschnitt wird beschrieben, wie Sie MicroVM-Images erstellen, konfigurieren, aktualisieren und verwalten.

Ein MicroVM-Image ist eine Ressource, die das Dateisystem und die Anwendungsumgebung einer MicroVM definiert. Das MicroVM-Image umfasst Ihre Laufzeitumgebung, Ihren Anwendungscode und unterstützende Programme wie Hintergrundprozesse und Observability-Agenten. Um ein MicroVM-Image zu erstellen, stellen Sie ein Zip-Paket bereit, das Ihre Anwendungsartefakte enthält `Dockerfile` und die Sie auf Amazon S3 hochladen.

Ihr `Dockerfile` definiert, wie Ihre Anwendung verpackt wird. Lambda erstellt Ihr Anwendungs-Container-Image, indem es `Dockerfile` auf einer Betriebssystemumgebung ausgeführt wird, die von einem Lambda-managed MicroVM-Basisimage bereitgestellt wird. MicroVM-Basis-Images werden unten im Abschnitt mit dem Titel — beschrieben. MicroVM-Basisimages

Sie können Ihre MicroVM-Basisimages aktualisieren, um den Anwendungscode oder die Konfiguration für Ihre MicroVMs zu aktualisieren. Bei jedem Update, das Sie auslösen, wird eine neue MicroVM-Image-Version erstellt.

Wie Lambda ein MicroVM-Image erstellt

Wenn Sie ein MicroVM-Image erstellen, führt Lambda Folgendes aus:

- Ruft Ihre verpackten Artefakte von Amazon S3 ab.
- Startet eine neue MicroVM aus dem Lambda-managed Basis-Image.
- Führt die Anweisungen in Ihrem aus. `Dockerfile`
- Startet Ihre Anwendung mithilfe der `CMD` Anweisung `ENTRYPOINT` oder.
- Wartet auf den Abschluss der Initialisierung, was durch Ihren Lifecycle-Hook signalisiert wird.
- Erfasst einen Snapshot des Festplatten- und Speicherstatus.

Sobald der Snapshot-Vorgang abgeschlossen ist, wechselt Ihr MicroVM-Image in den `CREATED` Status. Sie können dieses MicroVM-Image jetzt verwenden, um eine MicroVM zu erstellen, und jedes MicroVM-Image kann verwendet werden, um mehrere unabhängige MicroVMs zu erstellen. Eine MicroVM, die vom MicroVM-Image aus ausgeführt wird, wird direkt aus dem Snapshot-Status wieder aufgenommen, was für schnelle Startzeiten sorgt. Jedes MicroVM-Image kann verwendet werden, um mehrere MicroVMs auszuführen, bis zu dem für Ihr Konto verfügbaren Limit.

Eine schrittweise Anleitung zum Verpacken Ihres Codes und zum Erstellen Ihres ersten MicroVM-Images finden Sie unter. Erstellen Sie Ihre erste MicroVM

Dimensionierung von MicroVM

Lambda MicroVMS verwendet ein Baseline-Peak-Modell, das es überflüssig macht, jede Rechenumgebung für Spitzenaktivitäten richtig zu dimensionieren. Sie konfigurieren die grundlegenden Rechenressourcen für Ihre MicroVM. Bei Spitzenaktivität kann Ihre MicroVM automatisch vertikal bis auf das Vierfache des Ausgangswerts skalieren. Sie zahlen den Basistarif,

während Ihre MicroVM läuft, und zahlen nur für das, was Sie über dem Basispreis hinaus aktiv nutzen, und zwar pro Sekunde.

Sie legen den Basiswert fest, indem Sie den `memory` Parameter bei der Erstellung Ihres MicroVM-Images verwenden. vCPU skaliert proportional zum Arbeitsspeicher (2 GB = 1 vCPU). Die Standardbasislinie ist 2 GB/1 vCPU.

In der folgenden Tabelle sind die verfügbaren Größen aufgeführt:

Ausgangswert	Höhepunkt	Maximaler Speicherplatz
0,5 GB Speicher, 0,25 vCPU	2 GB Arbeitsspeicher, 1 vCPU	8 GB
1 GB Arbeitsspeicher, 0,5 vCPU	4 GB Arbeitsspeicher, 2 vCPU	8 GB
2 GB Arbeitsspeicher, 1 vCPU (Standard)	8 GB Arbeitsspeicher, 4 vCPU	8 GB
4 GB Speicher, 2 vCPU	16 GB Speicher, 8 vCPU	16 GB
8 GB Speicher, 4 vCPU	32 GB Speicher, 16 vCPU	32 GB

MicroVM-Basisimages

Ein MicroVM-Basisimage dient als Grundlage für Ihre MicroVM-Images. Lambda veröffentlicht ein MicroVM-Basis-Image, das das Amazon Linux 2023-Betriebssystem und die für den Betrieb von MicroVMs erforderlichen Servicekomponenten bereitstellt. Wenn Sie ein MicroVM-Image erstellen oder aktualisieren, startet Lambda eine neue MicroVM von diesem Basis-Image aus und führt Ihre `Dockerfile` Anweisungen in dieser Betriebssystemumgebung aus.

Lambda veröffentlicht regelmäßig neue Versionen der vom Service verwalteten MicroVM-Basis-Images, z. B. beim Anwenden von Sicherheitspatches, um das Betriebssystem oder die Servicekomponenten zu aktualisieren. Standardmäßig gilt die neueste Version eines vom Service verwalteten Basis-Images, wenn Sie creating/updating Ihre eigenen MicroVM-Images sind. Zur Problembehandlung oder zum Debuggen können Sie optional die Version der dienstverwalteten Images überschreiben, wenn Sie mithilfe des Parameters Ihr eigenes MicroVM-Image erstellen. `base-image-version`

Basis-Image-Versionen folgen einem Verfallszyklus:

- **AVAILABLE**— Aktuell, zur Verwendung empfohlen.
- **DEPRECATED**(60 Tage) — Eine neuere Version ist vorhanden. Sie können immer noch bauen und ausführen.
- **EXPIRING**(30 Tage) — Es können keine neuen Images erstellt werden. Bestehende Images können weiterhin ausgeführt werden.
- **EXPIRED**— Kann nicht erstellt oder ausgeführt werden. Erstellen Sie Ihr Image auf einer unterstützten Version neu.
- **RECALLED**— Aufgrund kritischer Sicherheitsprobleme sofort nicht verfügbar (selten).

Um auf dem neuesten Stand zu bleiben, achten Sie auf Benachrichtigungen über veraltete Versionen und erstellen Sie Ihre MicroVM-Images neu, wenn eine neue Basis-Image-Version veröffentlicht wird.

Beachten Sie, dass sich das MicroVM-Basisimage von dem Container-Basis-Image unterscheidet, das Sie in Ihren Dockerfiles angeben. Während ersteres die Betriebssystemumgebung für Ihre MicroVMs definiert, definiert letzteres, welches Basis-Container-Image verwendet werden soll, wenn Sie Ihre Anwendung für die Verwendung mit Lambda-MicroVMs verpacken. Weitere Informationen finden Sie im Abschnitt zu Container-Basisimages.

Verwenden Sie die folgenden APIs, um verfügbare verwaltete MicroVM-Basisimages und deren Versionen zu ermitteln:

```
# List all managed MicroVM base images
aws lambda-microvms list-managed-microvm-images

# List the versions of a specific managed MicroVM base image
aws lambda-microvms list-managed-microvm-image-versions \
  --image-identifier arn:aws:lambda:us-east-1:aws:microvm-image:al2023-1
```

Hooks zum Erstellen von MicroVM-Images

Lambda bietet MicroVM-Image-Build-Hooks, mit denen Sie die Richtigkeit der Anwendung überprüfen und die Leistung während der MicroVM-Image-Erstellung optimieren können. Hooks werden ausgeführt, bevor Lambda den Snapshot erstellt, der zur Initialisierung jeder MicroVM verwendet wird. Jeder Hook ist ein HTTP-Endpunkt, den Ihre Anwendung verfügbar macht und den Lambda während des Builds aufruft. Indem Sie auf diese Anfragen antworten, steuern und validieren Sie den

MicroVM-Image-Erstellungsprozess. Lambda verwendet HTTP-Statuscodes, um zu ermitteln, ob die Hooks erfolgreich abgeschlossen wurden.

> **Important**
>
> Wenn Sie Hooks konfigurieren, müssen Sie den Port angeben, auf dem Ihre Anwendung auf Hook-Anfragen wartet.

Hook	Pfad	Details	HTTP-Statuscodes	Zeitüberschreitung
/bereit	`/aws/lambda-microvms/runtime/v1/ready`	Wird während des MicroVM-Image-Builds aufgerufen, nachdem Ihre Anwendung durch `ENTRYPOINT` oder `CMD` gestartet wurde. Signalisiert, dass Ihre Anwendung bereit ist, einen Snapshot zu erstellen.	HTTP 503: Noch nicht bereit; Lambda versucht es bis zum Timeout erneut. HTTP 200: Die Initialisierung ist abgeschlossen; Lambda erstellt den Snapshot.	1—360 Sekunden (`readyTimeoutInSeconds`)
/validieren	`/aws/lambda-microvms/runtime/v1/validate`	Wird nach Abschluss des Builds auf einer neuen MicroVM aufgerufen, die mit dem erstellten Image gestartet wurde. Bestätigt, dass die Anwendung ordnungsgemäß funktioniert, wenn sie fortgesetzt wird.	HTTP 503: Die Validierung benötigt mehr Zeit, um abgeschlossen zu sein; Lambda versucht es erneut, bis das Zeitlimit überschritten ist. HTTP 200: Validierung bestanden.	1—3600 Sekunden (`validateTimeoutInSeconds`)

> **Important**
>
> Wenn Sie HTTP 503 zurückgeben, geben Sie es sofort zurück, anstatt die Anfrage offen zu halten, während Sie warten. Wenn der Timeout abläuft, während eine Anfrage offen gehalten wird, beendet Lambda den Build.

> **Note**
>
> Sie können auch den Hook /validate verwenden, um die Startzeit zu optimieren. Führen Sie dazu während der Validierung simulierte Nutzlasten aus. Auf diese Weise kann Lambda die Bereiche Ihres Snapshots verfolgen, auf die zugegriffen wurde, und deren Abruf beim MicroVM-Start optimieren.

Aktualisierung eines MicroVM-Images

Sie können ein vorhandenes MicroVM-Image aktualisieren, indem Sie die `update-microvm-image` API aufrufen. Jedes Update löst den Build einer neuen MicroVM-Image-Version aus. Normalerweise aktualisieren Sie ein MicroVM-Image auf:

- Neuen Anwendungscode bereitstellen — Verweisen Sie auf ein neues Code-Artefakt (eine neue Zip-Datei, die auf Amazon S3 hochgeladen wurde), um eine neue Version Ihrer Anwendung zu versenden.
- Zu einem neueren MicroVM-Basis-Image wechseln — Ändern Sie den ARN des MicroVM-Basisimages, um ein Upgrade auf neuere Versionen des Lambda MicroVM-Basis-Images durchzuführen. Weitere Informationen finden Sie unter MicroVM-Image-Patching und MicroVM-Basisimages.
- Ändern Sie die Build-Rolle — Aktualisieren Sie den Build-Rollen-ARN, wenn die Berechtigungen, die Lambda während des Builds benötigt, geändert werden, z. B. wenn Ihr Code-Artefakt in einen anderen Amazon S3-Bucket verschoben wird oder Sie beginnen, Daten aus einem privaten ECR-Repository abzurufen.
- Passen Sie die Laufzeitkonfiguration an — Ändern Sie Hooks, Umgebungsvariablen oder Funktionen, um neu zu konfigurieren, wie Ihr MicroVM-Image erstellt und ausgeführt wird.
- Beschreibung aktualisieren — Ändern Sie die MicroVM-Image-Beschreibung, um aufzuzeichnen, was sich in dieser Version geändert hat.

Der folgende CLI-Befehl zeigt, wie Sie ein MicroVM-Image aktualisieren können. Die `--build-role-arn` Parameter `--base-image-arn` und sind bei jedem `update-microvm-image` Aufruf erforderlich, der einen neuen Build auslöst, auch wenn Sie nur das Code-Artefakt ändern. Wenn Sie sie weglassen, erhalten Sie Folgendes: `ValidationException`

```
aws lambda-microvms update-microvm-image \
  --image-identifier arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-image \
  --code-artifact uri=s3://my-bucket/deployments/app-v2.zip \
  --base-image-arn arn:aws:lambda:us-east-1:aws:microvm-image:al2023-1 \
  --build-role-arn arn:aws:iam::123456789012:role/MicrovmBuildRole \
  --description "Updated with v2 application code"
```

Image-Status und Build-Status

Jedes Mal, wenn Sie ein MicroVM-Image erstellen oder aktualisieren, erstellt Lambda eine neue Version, die aus Ihrem Code-Artefakt und Ihrem Basis-Image erstellt wird. Ein MicroVM-Image kann im Laufe der Zeit viele Versionen haben, und Sie führen MicroVMs ab einer bestimmten Version aus.

Drei unabhängige Staaten verfolgen verschiedene Aspekte des Lebenszyklus:

- Image-Status — der gesamte Lebenszyklus der MicroVM-Image-Ressource (wird erstellt, ist einsatzbereit, wird aktualisiert, ist ausgefallen oder wird gelöscht).
- Versionsstatus — der Erstellungsfortschritt einer bestimmten Version (ausstehend, erstellt, erfolgreich oder fehlgeschlagen). Überprüfen Sie `stateReason` oder CloudWatch protokollieren Sie (`/aws/lambda-microvms/<image-name>`), um Einzelheiten zum Fehler anzuzeigen.
- Versionsaktivierung — ob eine erfolgreich erstellte Version MicroVMs ausführen darf. Lambda stellt neue Versionen `ACTIVE` automatisch ein; Sie können eine Version auf auf einstellen, um sie zu deaktivieren, ohne sie `INACTIVE` zu löschen.

Status	Mögliche Werte	Der Übergang wird gesteuert von
Status des Bildes	`CREATING`, `CREATED`, `CREATION_FAILED` , `UPDATING`, `UPDATED`, `UPDATE_FAILED` , `DELETING`, `DELETED`, `DELETION_FAILED`	Lambda (automatisch)

Status	Mögliche Werte	Der Übergang wird gesteuert von
Status der Version	`PENDING`, `IN_PROGRESS` , `SUCCESSFUL` , `FAILED`	Lambda (automatisch)
Aktivierung der Version	`ACTIVE`, `INACTIVE`	Du (`update-microvm-image-version --state`)

Um eine MicroVM von einer Version aus auszuführen, muss der Image-Status `CREATED` oder sein`UPDATED`, der Versionsstatus muss lauten`SUCCESSFUL`, und die Version muss `ACTIVE` lauten.

> **Note**
>
> Diese Staaten sind unabhängig. Ein Bild im `CREATED` Status kann eine Version enthalten, deren Status lautet`FAILED`.

```
# De-activate a version
aws lambda-microvms update-microvm-image-version \
  --image-identifier my-image \
  --image-version 1.0 \
  --state INACTIVE
```

Umgebungsvariablen

Umgebungsvariablen werden bei der Erstellung des MicroVM-Images über das `environmentVariables` Feld festgelegt (maximal 50 Variablen). Diese werden während des Snapshot-Erstellungsprozesses in den Container eingefügt. Sie können dynamisch festgelegte Nutzlasten übergeben, wenn Sie eine neue MicroVM ausführen. Weitere Informationen finden Sie im Abschnitt zum Ausführen Ihrer MicroVM.

Arbeiten mit Snapshots

Wenn AWS Lambda MicroVMS ein MicroVM-Image erstellt, erfasst es einen Firecracker-Snapshot Ihrer vollständig initialisierten Anwendung. Auf dieser Seite wird beschrieben, was ein Snapshot enthält, wie mit der Einzigartigkeit von MicroVMs umgegangen wird, die denselben Snapshot verwenden, und bewährte Methoden für Snapshot-kompatible Anwendungen.

Was ein Snapshot erfasst

Ein Firecracker-Snapshot erfasst den folgenden Status von Ihrer MicroVM zum Zeitpunkt des Abschlusses der Image-Erstellung:

- Speicherstatus der laufenden Prozesse — Alle von`ENTRYPOINT`/gestarteten Prozesse`CMD`, einschließlich Hintergrund-Daemons, Cron-Jobs und untergeordneter Prozesse.
- Festplattenstatus — Das Root-Dateisystem einschließlich aller während der Build-Phase geschriebenen Dateien (installierte Pakete, kompilierte Binärdateien, Konfigurationsdateien).
- Netzwerkverbindungen und Dateideskriptoren — Alle initialisierten Netzwerkverbindungen, Datei-Handles, Pipes und andere Deskriptoren.

Wenn Sie eine MicroVM ausführen, stellt Lambda sie aus diesem Snapshot wieder her. Ihre Anwendung wird genau dort fortgesetzt, wo der Build der MicroVM-Image-Version aufgehört hat.

Erwägungen zur Kompatibilität

Da alle MicroVMs, die mit derselben Image-Version gestartet wurden, den gleichen Ausgangszustand haben, sollten Sie beim Erstellen von Anwendungen mit Lambda-MicroVMs Folgendes beachten:

- Eindeutigkeit — Wenn Ihr Code während der Erstellungsphase der Image-Version eindeutigen Inhalt generiert (eindeutige IDs, Geheimnisse oder Entropie für Pseudozufälligkeit), wird dieser Inhalt von allen MicroVMs gemeinsam genutzt, die auf derselben Image-Version ausgeführt werden. Generieren Sie eindeutigen Inhalt nach dem Start einer neuen microVM, nicht während der Image-Erstellung. Verwenden Sie den `/run` Lifecycle-Hook, um zuvor generierte eindeutige Inhalte zurückzusetzen.
- Netzwerkverbindungen — Verbindungen, die während der Erstellungsphase der Image-Version hergestellt wurden, müssen möglicherweise erneut hergestellt werden, wenn eine MicroVM von der Image-Version aus ausgeführt wird. Überprüfen Sie den Verbindungsstatus und stellen Sie die Verbindungen bei Bedarf erneut her. In den meisten Fällen werden von AWS SDKs erstellte Verbindungen automatisch wiederhergestellt.

Umgang mit Eindeutigkeiten mit Schnappschüssen

Alle MicroVMs, die auf demselben Image ausgeführt werden, haben einen identischen Anfangszustand. Um die Einzigartigkeit aller MicroVMs zu wahren:

- Generieren Sie eindeutige IDs, Geheimnisse und Zufallswerte, nachdem eine MicroVM gestartet wurde und nicht während der Erstellung der Image-Version. Verwenden Sie den `/run` Lifecycle-Hook, um jeden eindeutigen Status zurückzusetzen.
- Verwenden Sie kryptografisch sichere Pseudozufallszahlengeneratoren (CSPRNGs) für Ihre Programmiersprache. Stellen Sie sicher, dass Ihre kryptografische Bibliothek dem Sprachstandard entspricht, ihre Entropie vom Gerät oder Gerät erhält. `/dev/random` `/dev/urandom` Dies ist das Standardverhalten für kryptografische Bibliotheken für gängige Programmiersprachen wie Java 11+ (SecureRandom Bibliothek), (Crypto.RandomBytes-Bibliothek), Python 3.12+ Node.js (Bibliothek) und Dotnet8+ (Bibliothek). Secrets.SystemRandom Cryptography.RandomNumberGenerator Software, die bei Verwendung mit MicroVM-Image-Snapshots Zufälligkeiten liest oder beibehält`/dev/random`. `/dev/urandom`
- Wenn Ihr Anwendungscode OpenSSL verwendet, verwenden Sie das AWS bereitgestellte Basis-Container-Image von, um Ihr MicroVM-Image `public.ecr.aws/lambda/microvms:al2023-minimal` zu erstellen. Dieses Image enthält eine AWS gepatchte Version von OpenSSL, die mit Snapshotting kompatibel ist. Wenn Sie Ihr eigenes Basis-Image mit OpenSSL verwenden möchten, verwenden Sie die gepatchte Version von OpenSSL (openssl-snapsafe-libs), die in der Amazon Linux 2023-Paketliste aufgeführt ist. https://docs.aws.amazon.com/linux/al2023/release-notes/all-packages.html

MicroVM-Image-Patching

Wenn ein neues MicroVM-Basisimage verfügbar ist, können Sie einen `update-microvm-image` Aufruf ausführen, um einen MicroVM-Image-Build mit den neuesten Patches auszulösen. Dabei können Sie entweder das Argument weglassen (für die neueste Version) oder das `base-image-version` Argument mit der neuesten Version angeben.

Container-Basisimages

Lambda MicroVMS führt Ihre Anwendung als Container in der MicroVM-Betriebssystemumgebung aus. Sie definieren diesen Container mit Ihrem`Dockerfile`, und die `FROM` Anweisung in Ihrem `Dockerfile` legt das Container-Basis-Image für Ihre Anwendung fest.

Sie können entweder mit dem Lambda-Basiscontainer-Image für Amazon Linux 2023 (`public.ecr.aws/lambda/microvms:al2023-minimal`) beginnen und darüber hinaus Ihre `Dockerfile` Anweisungen hinzufügen oder Ihr eigenes Basis-Container-Image verwenden. Wenn Sie Ihre eigenen Container-Images verwenden, überprüfen Sie die folgenden Anforderungen:

Voraussetzungen

- Das Container-Basisimage muss mit der Ziel-CPU-Architektur kompatibel sein.
- Für Container-Basis-Images aus privaten AWS ECR-Repositorys ist die Build-Rolle `ecr:GetAuthorizationToken` und `ecr:BatchGetImage` -Berechtigungen erforderlich.
- Das Container-Basis-Image muss auf einem Linux-Betriebssystem basieren.
- Auf die Container-Basis-Images muss von der Lambda-Build-Infrastruktur aus zugegriffen werden können (öffentliches Internet oder ein ECR-Repository im selben AWS Konto).
- Container-Basis-Images müssen Snapshot-kompatibel sein, siehe Anweisungen unten.

Snapshot-compatible Basis-Images

Da Lambda MicroVMS jede MicroVM von einem vorinitialisierten Snapshot aus startet, müssen die Basis-Images Snapshot-kompatibel sein. Wir empfehlen, den Abschnitt zur Erwägungen zur Kompatibilität Verwendung Ihrer eigenen Basis-Images mit Lambda MicroVMS zu lesen.

Verwenden eines privaten ECR-Images

Referenzieren Sie Ihr privates ECR-Container-Basis-Image in der `FROM` Anleitung Ihres: `Dockerfile`

```
FROM 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-base:latest
WORKDIR /app
COPY . .
CMD ["./my-app"]
```

Fügen Sie Ihrer Build-Rolle die folgenden Berechtigungen hinzu:

```
{
  "Effect": "Allow",
  "Action": [
    "ecr:GetAuthorizationToken",
    "ecr:BatchCheckLayerAvailability",
    "ecr:GetDownloadUrlForLayer",
    "ecr:BatchGetImage"
  ],
  "Resource": "*"
}
```

Funktionen des Betriebssystems

Standardmäßig werden Lambda-MicroVMs mit einem Standardsatz von Linux-Funktionen ausgeführt. Sie können erweiterte Linux-Funktionen gewähren, indem Sie das `additionalOsCapabilities` Feld verwenden, wenn Sie ein MicroVM-Image erstellen oder aktualisieren. Der einzige unterstützte Wert ist `["ALL"]`. Erweiterte Funktionen ermöglichen Operationen wie das Mounten von Dateisystemen, das Erstellen von Netzwerk-Namespaces oder das Ausführen von eBPF-Programmen. Funktionen werden innerhalb der Isolationsgrenze der virtuellen Maschine angewendet und wirken sich nicht auf den Host oder andere MicroVMs aus.

```
aws lambda-microvms create-microvm-image \
  --name my-network-tool \
  --code-artifact uri=s3://my-bucket/app.zip \
  --base-image-arn arn:aws:lambda:us-east-1:aws:microvm-image:al2023-1 \
  --build-role-arn arn:aws:iam::123456789012:role/BuildRole \
  --additional-os-capabilities '["ALL"]'
```

MicroVMs ausführen und verwenden

In diesem Abschnitt wird beschrieben, wie Sie MicroVMs starten, eine Verbindung zu Ihren laufenden Anwendungen herstellen, den MicroVM-Lebenszyklus verwalten und die Skalierung durchführen.

Eine MicroVM starten

Verwenden Sie den `run-microvm` Befehl, um eine neue MicroVM von einem bestimmten Image aus zu starten. Lambda stellt die erforderlichen Ressourcen bereit, erstellt einen dedizierten HTTPS-Endpunkt und startet Ihre Anwendung vom Image-Snapshot aus.

```
aws lambda-microvms run-microvm \
  --image-identifier arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-
image \
  --ingress-network-connectors "arn:aws:lambda:us-east-1:aws:network-connector:aws-
network-connector:ALL_INGRESS" \
  --egress-network-connectors "arn:aws:lambda:us-east-1:aws:network-connector:aws-
network-connector:INTERNET_EGRESS" \
  --idle-policy
 '{"autoResumeEnabled":true,"maxIdleDurationSeconds":900,"suspendedDurationSeconds":1800}'
 \
  --maximum-duration-in-seconds 14400
```

Eine MicroVM wird erstellt, wenn Sie aufrufen. `run-microvm` Jede MicroVM hat ihren eigenen dedizierten Endpunkt. Es gibt keinen Lastenausgleich zwischen MicroVMs von einem einzigen Endpunkt aus — jeder Endpunkt ist mit einer einzelnen MicroVM verknüpft.

Der einzige erforderliche Parameter ist `--image-identifier` (dies muss der ARN des MicroVM-Images sein). Alle anderen Parameter sind optional.

Hauptparameter

Parameter	Description
`--image-identifier`	(Erforderlich) Der ARN des auszuführenden MicroVM-Images.
`--image-version`	Die Version des MicroVM-Images, das ausgeführt werden soll. Standardmäßig wird die neueste aktive Version verwendet.
`--execution-role-arn`	Die IAM-Rolle, die der MicroVM Laufzeitberechtigungen für die Interaktion mit anderen Diensten gewährt. AWS
`--idle-policy`	Steuert das automatische Verhalten beim Anhalten und Wiederaufnehmen. Weitere Informationen zur Konfiguration von Richtlinien für den Leerlauf finden Sie im folgenden Abschnitt.
`--maximum-duration-in-seconds`	Die maximale Gesamtlebensdauer der MicroVM, einschlie ßlich der Dauer des Betriebs und des Ruhezustands. Lambda beendet die MicroVM, wenn diese Dauer überschritten wird. Bereich: 1—28.800 Sekunden (8 Stunden).
`--run-hook-payload`	Eine Zeichenkettennutzlast (max. 16 KB), die beim Start der MicroVM an den `/run` Lifecycle-Hook übermittelt wird.
`--logging`	Konfiguration protokollieren. Passen Sie die CloudWatch Protokollgruppe und den Stream an oder deaktivieren Sie die Protokollierung vollständig.
`--ingress-network-connectors`	Die ARN (s) der Ingress-Connectors, die eingehende HTTPS-Konnektivität ermöglichen.

Parameter	Description
`--egress-network-connectors`	Die ARN (s) der Ausgangskonnektoren für ausgehende Konnektivität (Internet oder VPC).

> **Note**
>
> Verwenden Sie den Connector, um die Eingangskonnektivität zu deaktivieren. Lambda-provided `NO_INGRESS` Weitere Informationen zu Netzwerkanschlüssen finden Sie unterthe section called “Netzwerk”.

Konfiguration der Richtlinie „Inaktiv“

Wenn diese Option aktiviert ist, steuert die Richtlinie für den Leerlauf die automatische Sperrung und Wiederaufnahme. Das Vorhandensein von Datenverkehr über den Endpunkt der MicroVM signalisiert Aktivität. Wenn während der konfigurierten Leerlaufdauer kein Datenverkehr eingeht, wird die MicroVM als inaktiv behandelt und angehalten.

Feld	Description
`autoResumeEnabled`	In diesem Fall nimmt die MicroVM automatisch den Betrieb wieder auf`true`, wenn der Datenverkehr an ihrem Endpunkt eintrifft, während er unterbrochen ist.
`maxIdleDurationSeconds`	Die Anzahl der Sekunden ohne Datenverkehr, nach denen die MicroVM angehalten wird. Maximum: 28.800 (8 Stunden).
`suspendedDurationSeconds`	Die Anzahl der Sekunden, in denen eine MicroVM im angehaltenen Zustand verbleibt, bevor Lambda sie beendet.

> **Note**
>
> Deaktivieren Sie für asynchrone Anwendungen, die keinen aktiven Datenverkehr über den Endpunkt senden oder empfangen, die automatische Sperrung oder konfigurieren Sie eine geeignete Leerlaufdauer.

Nutzlasten zur Laufzeit

Mit dem `runHookPayload` Parameter können Sie zur Laufzeit Konfigurationsdaten pro MicroVM (max. 16 KB Zeichenfolge) übergeben. Lambda übermittelt diese Nutzlast als Teil des Anforderungstextes an den Lifecycle-Hook. `/run` Lambda fügt das auch `microvmId` in den Anforderungstext ein.

Der `/run` Hook erhält einen JSON-Body mit der folgenden Struktur:

```
{
  "microvmId": "mvm-01234567-abcd-ef01-2345-6789abcdef01",
  "runHookPayload": "tenant-specific-string"
}
```

Verwenden Sie Laufzeit-Payloads, um eine Konfiguration bereitzustellen, die je nach MicroVM unterschiedlich ist — zum Beispiel Mandanten-IDs, Sitzungstoken, signierte URLs oder Secrets Manager-Pfade. Im Gegensatz zu Umgebungsvariablen (die auf Image-Ebene festgelegt werden und von allen MicroVMs aus diesem Image aus gemeinsam genutzt werden) ist die Run-Hook-Payload für jede MicroVM einzigartig.

```
aws lambda-microvms run-microvm \
  --image-identifier arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-image \
  --run-hook-payload 'tenant-specific-string'
```

Wenn Sie eine MicroVM nicht mehr benötigen, beenden Sie sie, um alle Ladevorgänge zu beenden. Detaillierte Anweisungen finden Sie unter the section called “Beenden einer MicroVM”.

Verbindung zu einer MicroVM herstellen

Jede MicroVM erhält eine eindeutige öffentliche HTTPS-Endpunkt-URL, die beim Aufruf zugewiesen wird. `run-microvm` Über diese URL stellen Sie eine Verbindung zu Ihrer Anwendung her, die in der MicroVM ausgeführt wird.

Authentifizierung

Für alle Anfragen an einen MicroVM-Endpunkt ist ein JWE-Authentifizierungstoken erforderlich. Es gibt keine unauthentifizierte Zugriffsoption. Generieren Sie ein Token mit: `create-microvm-auth-token`

```
aws lambda-microvms create-microvm-auth-token \
  --microvm-identifier microvm-id \
  --expiration-in-minutes 30 \
  --allowed-ports '[{"allPorts":{}}]'
```

Token sind auf bestimmte Ports beschränkt und haben ein konfigurierbares Ablaufdatum. Sie können den Zugriff auf einen einzelnen Port, einen Portbereich oder alle Ports einschränken:

```
{ "port": number }
{ "range": { "startPort": N, "endPort": N } }
{ "allPorts": {} }
```

Port-Routing

Standardmäßig leitet Lambda eingehenden Datenverkehr an Port 8080 in Ihrer MicroVM weiter. Um an einen anderen Port weiterzuleiten, fügen Sie den `X-aws-proxy-port` Header in Ihre Anfrage ein. Der Zielport muss innerhalb des im Authentifizierungstoken `allowedPorts` definierten Ports liegen.

Protokolle

Lambda MicroVMS unterstützt HTTP/2 WebSockets, gRPC und SSE über die Endpunkt-URL.

Übergeben Sie bei WebSocket Verbindungen das Authentifizierungstoken und den Zielport über Unterprotokolle:

```
// JavaScript WebSocket example
const protocols = [
  "lambda-microvms",                              // Required base protocol
  "lambda-microvms.authentication.<auth-token>",  // Auth token
  "lambda-microvms.port.9000"                     // Target port
];
const ws = new WebSocket('wss://<microvm-endpoint>/path', protocols);
```

Lambda entfernt MicroVM-specific Unterprotokolle aus der Anfrage, bevor sie an Ihre Anwendung weitergeleitet wird.

SDK-Beispiele

Die folgenden Beispiele zeigen, wie Sie eine MicroVM ausführen und mithilfe der AWS SDKs eine Verbindung zu ihr herstellen.

Python

Example Beispiel — Eine MicroVM ausführen und eine Verbindung mit boto3 herstellen

```
import boto3, requests
client = boto3.client("lambda-microvms")
run_resp = client.run_microvm(
    imageIdentifier="arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-
image",
    idlePolicy={"autoResumeEnabled": True, "maxIdleDurationSeconds": 900,
 "suspendedDurationSeconds": 300}
)
microvm_id = run_resp["microvmId"]
endpoint = run_resp["endpoint"]
print(f"MicroVM {microvm_id} running at {endpoint}")
token_resp = client.create_microvm_auth_token(
    microvmIdentifier=microvm_id, expirationInMinutes=30, allowedPorts=[{"allPorts":
 {}}]
)
token = token_resp["authToken"]["X-aws-proxy-auth"]
resp = requests.get(f"https://{endpoint}/health", headers={"X-aws-proxy-auth":
 token})
print(resp.status_code, resp.json())
```

Node.js

Example Beispiel — Eine MicroVM ausführen und eine Verbindung mit dem herstellen AWS SDK für JavaScript

```
import { LambdaMicrovmsClient, RunMicrovmCommand, CreateMicrovmAuthTokenCommand }
 from "@aws-sdk/client-lambda-microvms";
const client = new LambdaMicrovmsClient({});
const { microvmId, endpoint } = await client.send(new RunMicrovmCommand({
  imageIdentifier: "arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-
image",
  idlePolicy: { autoResumeEnabled: true, maxIdleDurationSeconds: 900,
 suspendedDurationSeconds: 300 }
}));
const { authToken } = await client.send(new CreateMicrovmAuthTokenCommand({
  microvmIdentifier: microvmId, expirationInMinutes: 30, allowedPorts: [{ allPorts:
 {} }]
}));
```

```
const resp = await fetch(`https://${endpoint}/health`, {
  headers: { "X-aws-proxy-auth": authToken["X-aws-proxy-auth"] }
});
console.log(await resp.json());
```

Anfragen senden

Bash

Example Beispiel — Senden einer Anfrage mit cURL

```
curl 'https://<microvm-endpoint>' \
  -H 'X-aws-proxy-auth: <TOKEN>' \
  -H 'X-aws-proxy-port: 8080'
```

Python

Example Beispiel — Senden einer Anfrage mit der Anforderungsbibliothek

```
import requests
response = requests.get('https://<microvm-endpoint>', headers={'X-aws-proxy-auth':
 '<TOKEN>'})
print(response.text)
```

Node.js

Example Beispiel — Senden einer Anfrage mit Fetch

```
const response = await fetch('https://<microvm-endpoint>', {
  headers: { 'X-aws-proxy-auth': '<TOKEN>', 'X-aws-proxy-port': '8080' }
});
console.log(await response.text());
```

Lebenszyklus-Hooks

Mithilfe von Lifecycle-Hooks können Sie benutzerdefinierte Logik an wichtigen Punkten im MicroVM-Lebenszyklus ausführen — wenn der MicroVM-Lebenszyklus beginnt, unterbrochen, fortgesetzt oder

beendet wird. Verwenden Sie Hooks, um den Status pro Mandant zu initialisieren, Daten vor dem Aussetzen zu löschen, Anmeldeinformationen bei Wiederaufnahme zu aktualisieren oder Ressourcen vor dem Beenden zu bereinigen.

Jeder Hook ist ein HTTP-Endpunkt, den Ihre Anwendung verfügbar macht. Lambda sendet beim entsprechenden Lebenszyklusereignis eine POST-Anforderung an den Hook. Hooks hören den Pfad `/aws/lambda-microvms/runtime/v1/<hook-name>` auf dem Port ab, den Sie konfigurieren.

Ihre MicroVM beginnt, externen Datenverkehr zu empfangen, nachdem der `/run` Hook HTTP 200 zurückgegeben hat. Bis dahin leitet der Endpunkt keine Anfragen an Ihre Anwendung weiter.

Haken	Wenn aufgerufen	Zweck
`/aws/lambda-microvms/runtime/v1/run`	Nachdem MicroVM mit einem Snapshot gestartet wurde	Initialisieren Sie den Status pro Mandant, setzen Sie eindeutige Werte zurück und führen Sie Integritätsprüfungen durch. Der Verkehr beginnt, nachdem dieser Hook zurückgekehrt ist.
`/aws/lambda-microvms/runtime/v1/resume`	Nachdem MicroVM aus dem angehaltenen Zustand wieder aufgenommen wurde	Re-establish Netzwerkverbindungen, Anmeldeinformationen aktualisieren, Status überprüfen. Die MicroVM bleibt im `SUSPENDED` Status, während dieser Hook ausgeführt wird; sie wechselt in diesen Zustand, `RUNNING` nachdem der Hook zurückgekehrt ist.
`/aws/lambda-microvms/runtime/v1/suspend`	Bevor MicroVM den Vorgang unterbricht	Löscht ausstehende Schreibvorgänge, schließt Verbindungen, gibt Ressourcen frei.
`/aws/lambda-microvms/runtime/v1/terminate`	Bevor MicroVM beendet wird	Daten löschen, externe Systeme benachrichtigen, bereinigen. Dieser Hook läuft nur bei endgültiger Kündigung, nicht beim Suspendieren. Verwenden Sie den `/suspend` Hook für die Bereinigung vor dem Aussetzen.

Informationen zu Hooks, die während der Image-Erstellung (`/readyund/validate`) ausgeführt werden, finden Sie unter. Hooks zum Erstellen von MicroVM-Images

OpenAPI-Spezifikation:

```
{
  "openapi": "3.0.2",
  "info": {
    "title": "Lambda MicroVMs Application Hook Interface",
    "version": "2025-12-03"
  },
  "paths": {
    "/ready": {
      "post": {
        "description": "Called by Lambda during MicroVM image creation to determine if
 the application has initialized.",
        "operationId": "Ready",
        "responses": {
          "200": { "description": "Successful invocation." },
          "503": { "description": "Application is not yet ready. Lambda retries until
 timeout." }
        }
      }
    },
    "/resume": {
      "post": {
        "description": "Called by Lambda when resuming a MicroVM that is in the
 SUSPENDED state.",
        "operationId": "Resume",
        "responses": {
          "200": { "description": "Successful invocation." }
        }
      }
    },
    "/run": {
      "post": {
        "description": "Called by Lambda when a new MicroVM is run from a MicroVM
 image.",
        "operationId": "Run",
        "requestBody": {
          "content": {
            "application/json": {
              "schema": { "$ref": "#/components/schemas/RunRequestContent" }
            }
          }
        },
        "responses": {
```

```
          "200": { "description": "Successful invocation." }
        }
      }
    },
    "/suspend": {
      "post": {
        "description": "Called by Lambda when suspending a MicroVM.",
        "operationId": "Suspend",
        "responses": {
          "200": { "description": "Successful invocation." }
        }
      }
    },
    "/terminate": {
      "post": {
        "description": "Called by Lambda when terminating a MicroVM, before resources
are released.",
        "operationId": "Terminate",
        "responses": {
          "200": { "description": "Successful invocation." }
        }
      }
    },
    "/validate": {
      "post": {
        "description": "Called by Lambda when running a MicroVM to validate the
image build. Use this hook to perform tests that validate your application behaves
correctly when running. Lambda also samples the portions of the image that are used
when handling this request, allowing Lambda to prefetch those portions of the image to
reduce latency at run time.",
        "operationId": "Validate",
        "responses": {
          "200": { "description": "Successful invocation." },
          "503": { "description": "Validation in progress. Lambda retries until
timeout." }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "RunRequestContent": {
        "type": "object",
        "properties": {
```

```
          "microvmId": {
            "type": "string",
            "description": "The MicroVM identifier."
          },
          "runHookPayload": {
            "type": "string",
            "description": "Run hook payload provided to RunMicrovm."
          }
        }
      }
    }
  },
  "servers": [
    { "url": "/aws/lambda-microvms/runtime/v1" }
  ]
}
```

Aussetzen und Fortsetzen von MicroVMs

Setzen Sie MicroVMs aus, um die Kosten zu senken und gleichzeitig den Anwendungsstatus beizubehalten. Während der Ausführung zahlen Sie Rechengebühren. Während der Sperrung zahlen Sie nur Snapshot-Speichergebühren.

Wie kann ich aussetzen

Es gibt zwei Möglichkeiten, eine MicroVM zu suspendieren:

1. Richtlinie für Leerlauf (automatisch) — Konfigurieren Sie diese Richtlinie `maxIdleDurationSeconds` in der Leerlaufrichtlinie. Wenn während dieser Dauer kein Datenverkehr am MicroVM-Endpunkt ankommt, unterbricht Lambda die MicroVM automatisch.
2. API-Aufruf (explizit) — Aufruf `suspend-microvm` zum sofortigen Aussetzen:

```
aws lambda-microvms suspend-microvm --microvm-identifier microvm-id
```

Der /suspend-Hook

Vor dem Suspendieren ruft Lambda Ihren Hook auf. `/suspend` Verwenden Sie es, um ausstehende Schreibvorgänge zu löschen, Netzwerkverbindungen zu schließen und Ressourcen freizugeben, die nicht über die Suspend-Grenze hinaus bestehen dürfen.

Verhalten wieder aufnehmen

Wenn eine MicroVM wieder aufgenommen wird (durch einen API-Aufruf oder automatische Wiederaufnahme), stellt Lambda den Speicher- und Festplattenstatus vom Suspend-Checkpoint aus wieder her. Die MicroVM bleibt im Status, während der Hook ausgeführt wird. `SUSPENDED` / `resume` Nachdem der Hook HTTP 200 zurückgegeben hat, wechselt die MicroVM zum Datenverkehr `RUNNING` und beginnt, diesen zu empfangen.

Verwenden Sie den `/resume` Hook, um Anmeldeinformationen zu aktualisieren, Netzwerkverbindungen wiederherzustellen und den Status zu überprüfen.

```
aws lambda-microvms resume-microvm --microvm-identifier microvm-id
```

Auto-resume

Wenn `autoResumeEnabled=true` der Datenverkehr den Endpunkt einer gesperrten MicroVM erreicht, nimmt Lambda die MicroVM automatisch wieder auf. Lambda speichert die eingehende Anfrage, bis die Wiederaufnahme abgeschlossen ist (einschließlich des `/resume` Hooks), und übermittelt sie dann an Ihre Anwendung.

Der Lebenslauf erhöht die Latenz bei der ersten Anfrage. Die Dauer hängt von der Größe des wiederherzustellenden Suspende-Zustands und der Dauer Ihres `/resume` Hooks ab.

Wenn die Wiederaufnahme nicht erfolgreich ist, gibt Lambda 502 Bad Gateway an den Aufrufer zurück.

> **Note**
>
> Auto-resume fügt nur der ersten Anfrage nach dem Aussetzen Latenz hinzu. Nachfolgende Anfragen, während die MicroVM läuft, sind davon nicht betroffen.

Skalierung und Parallelität

Sie erstellen neue MicroVMs, indem Sie anrufen. `run-microvm` Jede MicroVM hat ihren eigenen dedizierten Endpunkt. Es gibt keinen Lastenausgleich zwischen MicroVMs von einem einzigen Endpunkt aus.

Account-level Kapazität — Ihr Konto hat ein Kontingent für den gesamten Arbeitsspeicher, der all Ihren MicroVMs in der Region `RUNNING` oder im `SUSPENDED` Bundesstaat zugewiesen werden kann,

und Sie können vertikal auf das Vierfache dieses Kontingents skalieren. Um eine Erhöhung des Kontingents anzufordern, besuchen Sie die Service Quota-Konsole und suchen Sie nach Lambda MicroVMs.

Kostenmodell:

- Für das Ausführen von MicroVMs fallen Rechenkosten an.
- Für gesperrte MicroVMs fallen Snapshot-Speichergebühren an, aber keine Rechenkosten.
- Für gekündigte MicroVMs fallen keine Gebühren an.

Strategien für das Kapazitätsmanagement:

- MicroVMs im Leerlauf aussetzen — Konfigurieren Sie Richtlinien für den Leerlauf, um MicroVMs, die keinen Datenverkehr empfangen, automatisch zu sperren.
- Nicht mehr benötigte MicroVMs beenden — Verwenden Sie diese Option, um sie nach einer maximalen Sperrdauer automatisch `suspendedDurationSeconds` zu beenden, oder rufen Sie explizit an. `terminate-microvm`
- Right-size Richtlinien für inaktive Nutzung — Diese Richtlinien werden auf `maxIdleDurationSeconds` der Grundlage Ihrer Verkehrsmuster festgelegt. Durch kürzere Leerlaufzeiten werden Kapazitäten schneller freigesetzt.

Beenden einer MicroVM

Beenden Sie eine MicroVM, wenn sie nicht mehr benötigt wird. Bei einer Kündigung werden alle Rechenressourcen freigegeben und alle Gebühren werden gestoppt.

Vor der Freigabe von Ressourcen ruft Lambda Ihren `/terminate` Hook auf. Verwenden Sie es, um ausstehende Daten zu löschen oder externe Systeme zu benachrichtigen. Der `/terminate` Hook wird nur beim endgültigen Beenden ausgeführt — er läuft nicht, wenn eine MicroVM angehalten ist.

```
aws lambda-microvms terminate-microvm --microvm-identifier microvm-id
```

MicroVMs auflisten

Listet alle MicroVMs in Ihrem Konto auf, optional nach Bildern gefiltert:

```
aws lambda-microvms list-microvms
```

```
# Filter by image
aws lambda-microvms list-microvms --image-identifier my-image --image-version 1.0
```

Fehlerbehandlung

Fehler ausführen

In der folgenden Tabelle sind die häufigsten Fehler aufgeführt, die von der `run-microvm` API zurückgegeben werden:

Fehler	Ursache	Lösung
`ServiceQuotaExceededException`	Das Konto hat sein Speicherkontingent für gleichzeitige microVMs erreicht.	Beenden Sie inaktive MicroVMs oder fordern Sie eine Erhöhung des Kontingents an.
`ResourceNotFoundException`	Das angegebene Image ist nicht vorhanden oder befindet sich nicht im `CREATED` Status.	Überprüfen Sie die Image-ID und bestätigen Sie, dass der Build abgeschlossen ist.
`ValidationException`	Ein oder mehrere Anforderungsparameter sind ungültig.	Überprüfen Sie die Richtlinienwerte für den Leerlauf, das Image-Identifikationsformat und die Connector-ARNs.
`ThrottlingException`	Das API-Ratenlimit für diesen Vorgang wurde überschritten.	Implementieren Sie exponentielles Backoff mit Jitter.

Versuchen Sie es erneut mit der Strategie

Verwenden Sie für vorübergehende Fehler (`ThrottlingException`,`InternalServerException`) den exponentiellen Backoff:

```
import time, random
```

```
def run_with_retry(client, params, max_retries=5):
    for attempt in range(max_retries):
        try:
            return client.run_microvm(**params)
        except client.exceptions.ThrottlingException:
            delay = (2 ** attempt) + random.uniform(0, 1)
            time.sleep(delay)
    raise Exception("Max retries exceeded")
```

Informationen zu unterstützten Serviceintegrationen mit Lambda Managed Instances finden Sie unter. the section called “Integrationen”

Netzwerk

Sie konfigurieren den Netzwerkzugriff für Ihre AWS Lambda microVMs, indem Sie Ihrer microVM zur Laufzeit Network Connector-Ressourcen zuordnen. Netzwerkanschlüsse werden beim Aufrufen angegeben `run-microvm` und können nicht geändert werden, während eine MicroVM läuft.

-Übersicht

Jede MicroVM kann über unabhängige Netzwerkkonfigurationen für den Eingang (eingehend) und den Ausgang (ausgehend) verfügen:

- Eingangsnetzwerkanschlüsse ermöglichen eingehende Konnektivität. Clients stellen eine Verbindung zu einem vom Service verwalteten HTTPS-Endpunkt her, und Lambda leitet den Datenverkehr an Ports weiter, die Sie in der MicroVM konfigurieren. Ingress-Connectors werden AWS verwaltet — Sie referenzieren sie per ARN, wenn Sie eine MicroVM ausführen.
- Ausgehende Netzwerkanschlüsse ermöglichen ausgehenden Datenverkehr. Standardmäßig haben MicroVMs öffentlichen Internetzugang. Sie können stattdessen einen vom Kunden verwalteten VPC-Ausgangsconnector erstellen, um ausgehenden Datenverkehr über Ihre VPC weiterzuleiten.

Ein einziger Connector kann für viele MicroVMs wiederverwendet werden — dies ist das beabsichtigte Verwendungsmuster.

Eingehende Konnektivität

Jede Lambda MicroVM ist über eine eindeutige HTTPS-Endpunkt-URL erreichbar, die Ihnen bei einem Anruf zugewiesen wird. `run-microvm` Clients senden Anfragen über HTTPS an diesen

Endpunkt. Lambda leitet jede Anfrage an einen Port in Ihrer MicroVM weiter, wo Ihre Anwendung sie empfängt.

Standardmäßig werden am Endpunkt empfangene Anfragen an Port 8080 innerhalb der MicroVM weitergeleitet. Informationen zur Weiterleitung an einen anderen Port finden Sie unter. the section called “Port-Routing”

Die folgenden Protokolle werden auf dem Eingangsendpunkt unterstützt:

- HTTP/1.1
- HTTP/2
- WebSockets
- gRPC
- Server-Sent Ereignisse (SSE)

> **Note**
>
> Der Datenverkehr zwischen Ihrem Client und dem MicroVM-Endpunkt wird immer mit TLS verschlüsselt. Ihre Anwendung kann Anfragen intern entweder über HTTP oder HTTPS bearbeiten.

Port-Routing

Lambda wählt den Zielport in Ihrer MicroVM in der folgenden Prioritätsreihenfolge aus:

1. **`X-aws-proxy-port`**Header — Fügen Sie bei Standard-HTTP-Anfragen diesen Header mit der Zielportnummer hinzu.
2. WebSocket Unterprotokoll — Wenn Ihr WebSocket Client keine benutzerdefinierten Header einrichten kann, geben Sie den Zielport als Unterprotokoll mit dem Namen an`lambda-microvms.port.`*N*, wobei die Portnummer *N* steht. Sie geben Unterprotokolle an, wenn Sie die Verbindung öffnen. WebSocket Ein Beispiel finden Sie unter the section called “Protokolle”.
3. Standard (8080) — Wenn keines von beiden angegeben ist, wird die Route zu Port 8080 angefordert.

> **Important**
>
> Der Zielport muss innerhalb des im `allowedPorts` Authentifizierungstoken definierten Ports liegen. Anfragen an nicht autorisierte Ports erhalten die Antwort 403 Forbidden.

Authentifizierung

Alle Anfragen an einen MicroVM-Endpunkt erfordern ein gültiges Authentifizierungstoken im `X-aws-proxy-auth` Header. Sie generieren Token mit`create-microvm-auth-token`. Jedes Token ist eine verschlüsselte JWE-Zeichenfolge (JSON Web Encryption) mit folgendem Gültigkeitsbereich:

- Eine bestimmte MicroVM (identifiziert durch ID).
- Eine Reihe zulässiger Ports (einzelner Port, Bereich oder alle Ports).
- Eine Ablaufzeit (konfiguriert bei der Token-Erstellung).

Das folgende Beispiel erstellt ein Token und verwendet es, um eine authentifizierte Anfrage zu senden:

```
aws lambda-microvms create-microvm-auth-token \
  --microvm-identifier microvm-id \
  --expiration-in-minutes 30 \
  --allowed-ports '[{"port":8080}]'
```

```
curl 'https://microvm-endpoint' \
  -H 'X-aws-proxy-auth: TOKEN' \
  -H 'X-aws-proxy-port: 8080'
```

Eine vollständige Anleitung zum Erstellen von Token und zum Herstellen einer Verbindung zu einer MicroVM, einschließlich WebSocket Verbindungen, finden Sie unter. the section called “Verbindung zu einer MicroVM herstellen”

Fehlermeldungen

Die folgenden HTTP-Statuscodes werden vom MicroVM-Endpunkt zurückgegeben, wenn er eine Anfrage nicht verarbeiten oder an Ihre Anwendung senden kann. Diese Antworten kommen vom Endpunkt, nicht von Ihrer Anwendung.

Code	Status	Ursache und Lösung
400	Inkorrekte Anfrage	Fehlerhafte Anfrage oder ein ungültiger Port-Header oder ein ungültiges WebSocket Unterprotokoll. Überprüfen Sie das Format.
403	Forbidden	Fehlendes, abgelaufenes oder ungültiges Token oder der angeforderte Port ist nicht im Token enthalten`allowedPorts` . Generieren Sie ein neues Token oder verwenden Sie einen zulässigen Port.
429	Zu viele Anfragen	Das Ratenlimit wurde überschritten (auf Kontoebene oder pro microVM). Versuchen Sie es erneut mit exponentiellem Backoff.
500	Internal Server Error	Es ist ein interner Fehler aufgetreten. Wiederholen Sie die Anforderung.
502	Bad Gateway	Die Anwendung reagiert nicht oder die automatische Wiederaufnahme war innerhalb der maximalen Anzahl von Wiederholungsversuchen nicht erfolgreich. Siehe the section called “Auto-resume”.

Anfordern von Headern

Der `X-aws-proxy-*` Header-Namespace wird von Lambda für Anforderungsmetadaten wie das Authentifizierungstoken (`X-aws-proxy-auth`) und den Zielport (`X-aws-proxy-port`) reserviert. Lambda entfernt `X-aws-proxy-*` Header, bevor die Anfrage an Ihre Anwendung weitergeleitet wird.

Request/response Bandbreite

Jede Lambda MicroVM hat eine request/response Bandbreite, die linear mit ihrer Größe skaliert. Diese Bandbreite gilt für den gesamten Datenverkehr über den MicroVM-Endpunkt, sowohl für eingehende Anfragen als auch für ausgehende Antworten.

Größe der MicroVM (Basiswert)	Max. Bandbreite
0,5 GB, 0,25 vCPU	1 MB/s (8 Mbit/s)

Größe der MicroVM (Basiswert)	Max. Bandbreite
1 GB, 0,5 vCPU	2 MB/s (16 Mbit/s)
2 GB, 1 vCPU	4 MB/s (32 Mbit/s)
4 GB, 2 vCPU	8 MB/s (64 Mbit/s)
8 GB, 4 vCPU	16 MB/s (128 Mbit/s)

Wenn Sie aufgrund der Netzwerksättigung eine erhöhte Latenz bei Anfragen feststellen, reduzieren Sie entweder die Parallelität Ihrer Anfragen oder die Größe der Nutzlast, oder wählen Sie eine größere MicroVM-Größe, um die verfügbare Bandbreite zu erhöhen.

HTTP/2 Unterstützung

Lambda MicroVMS unterstützt HTTP/2 am eingehenden Endpunkt. Lambda handelt das Protokoll während des TLS-Handshakes über ALPN (Application-Layer Protocol Negotiation) aus, wobei es bevorzugt HTTP/2 wird und darauf zurückgreift. HTTP/1.1 Ein HTTP/2-capable Client verwendet es automatisch.

So verwenden Sie es HTTP/2 zwischen dem Endpunkt und Ihrer Anwendung innerhalb der MicroVM:

- Ihre Anwendung verwendet TLS — Lambda verhandelt HTTP/2 mit Ihrer Anwendung über ALPN und greift darauf zurück, HTTP/1.1 falls dies HTTP/2 nicht unterstützt wird.
- Ihre Anwendung verwendet HTTP im Klartext-Format — Fügen Sie den `X-aws-proxy-force-h2: true` Header in Ihre Anfrage ein, der für die Verbindung zu Ihrer Anwendung verwendet HTTP/2 werden soll.

Ausgehende Konnektivität

Standardmäßig haben Lambda MicroVMs öffentlichen Internetzugang auf dem Ausgangspfad. Um MicroVMs mit Ressourcen in Ihren privaten VPCs — wie RDS ElastiCache, internen APIs und lokalen Systemen über Direct Connect oder VPN — zu verbinden, erstellen Sie einen Lambda Network Connector mit Ihrer VPC-Konfiguration.

Wenn Sie VPC-Egress verwenden, unterliegt ausgehender Datenverkehr Sicherheitsgruppenregeln und Netzwerk-ACLs, die den Verkehr in Ihrer VPC regeln.

Arbeiten mit ausgehenden Netzwerkverbindern

Ausgangsnetzwerkanschlüsse leiten ausgehenden Datenverkehr von Ihrer MicroVM über Ihre VPC weiter. Sie erstellen einmal einen Connector und verweisen dann per ARN darauf, wenn Sie MicroVMS über den `run-microvm` Befehl starten.

Voraussetzungen

Bevor Sie einen Netzwerkconnector erstellen, benötigen Sie eine IAM-Rolle, die es Lambda ermöglicht, Elastic Network Interfaces (ENIs) in Ihrer VPC zu erstellen. Die Rolle erfordert die folgenden Berechtigungen:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CreateENI",
      "Effect": "Allow",
      "Action": "ec2:CreateNetworkInterface",
      "Resource": [
        "arn:aws:ec2:*:*:network-interface/*",
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Sid": "TagENI",
      "Effect": "Allow",
      "Action": "ec2:CreateTags",
      "Resource": "arn:aws:ec2:*:*:network-interface/*",
      "Condition": {
        "StringEquals": {
          "ec2:ManagedResourceOperator": "network-connectors.lambda.amazonaws.com"
        }
      }
    }
  ]
}
```

Einen Netzwerkkonnektor erstellen

Erstellen Sie einen Connector, indem Sie Ihre VPC-Subnetze, Sicherheitsgruppen und Ihr Netzwerkprotokoll angeben (`IPv4`oder`DualStack`):

```
aws lambda-core create-network-connector \
  --name my-connector \
  --configuration '{
    "VpcEgressConfiguration": {
      "SubnetIds": ["subnet-xxx"],
      "SecurityGroupIds": ["sg-xxx"],
      "NetworkProtocol": "IPv4",
      "AssociatedComputeResourceTypes": ["MicroVm"]
    }
  }' \
  --operator-role arn:aws:iam::123456789012:role/NetworkConnectorOperatorRole
```

Status des Netzwerkkonnektors

Ein Connector muss sich im `ACTIVE` Status befinden, bevor Sie ihn referenzieren können`run-microvm`.

Status	Description
`PENDING`	Der Connector wird erstellt (die zugrunde liegenden ENIs werden bereitgestellt).
`ACTIVE`	Der Connector ist einsatzbereit.
`INACTIVE`	Der Connector ist vorübergehend inaktiv.
`FAILED`	Die Bereitstellung oder Aktualisierung ist fehlgeschlagen. Überprüfen Sie `StateReason` .
`DELETING`	Der Connector wird gelöscht; ENIs werden bereinigt.
`DELETE_FAILED`	Der Löschvorgang ist fehlgeschlagen.

Versionierung des Netzwerkkonnektors

Die Versionierung gilt für VPC-Ausgangsnetzwerkanschlüsse (Connectortyp). `VPC_EGRESS` Ein VPC-Ausgangsconnector ist versioniert: Bei jedem erfolgreichen `update-network-connector` Aufruf wird eine neue Version erstellt, und frühere Versionen bleiben verfügbar.

Wenn Sie eine MicroVM ausführen, verweisen Sie per ARN auf einen VPC-Ausgangsconnector. `--egress-network-connectors` Sie können eine der beiden Formen verwenden:

- Unqualifizierter ARN — löst auf die neueste Version des Connectors auf. Eine MicroVM nimmt Konfigurationsänderungen automatisch auf. `arn:aws:lambda:`*`us-east-1`*`:`*`123456789012`*`:network-connector:`*`nc-id`*
- Qualifizierter ARN — verbindet die MicroVM mit einer bestimmten Version. Verwenden Sie diese Option, um eine microVM zum Testen, zur Fehlerbehebung oder zum Rollback in einer bekannten Konfiguration zu speichern. `arn:aws:lambda:`*`us-east-1`*`:`*`123456789012`*`:network-connector:`*`nc-id`*`:`*`version-number`*

Eine MicroVM wird nur dann auf eine neue Version umgestellt, wenn Sie den ARN ändern, auf den sie verweist, sodass Sie neue Versionen nach Ihrem eigenen Zeitplan übernehmen können.

Beispiel für einen `run-microvm` Aufruf, der eine bestimmte Connector-Version anheftet:

```
aws lambda-microvms run-microvm \
  --image-identifier arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-image \
  --egress-network-connectors arn:aws:lambda:us-east-1:123456789012:network-connector:nc-id:version-number \
  --idle-policy
 '{"maxIdleDurationSeconds":900,"suspendedDurationSeconds":1800,"autoResumeEnabled":false}'
```

Betrieb einer MicroVM mit einem Netzwerkanschluss

Verweisen Sie auf den Connector-ARN, wenn Sie eine MicroVM ausführen:

```
aws lambda-microvms run-microvm \
  --image-identifier arn:aws:lambda:us-east-1:123456789012:microvm-image:my-microvm-image \
  --egress-network-connectors connector-arn \
  --idle-policy
 '{"maxIdleDurationSeconds":900,"suspendedDurationSeconds":1800,"autoResumeEnabled":false}'
```

> **Note**
>
> Bevor Sie einen Connector aktualisieren oder löschen, stellen Sie sicher, dass alle MicroVMs, die ihn verwenden, beendet wurden. Das Ändern eines Connectors, der aktiv verwendet wird, kann zu Netzwerkverbindungsproblemen beim Ausführen von MicroVMs führen.

Verwendung von Lambda MicroVMS mit Schnittstellen-VPC-Endpunkten (AWS PrivateLink)

Sie können private Konnektivität über das AWS Netzwerk zwischen Ihren VPC-Ressourcen und Lambda MicroVMs verwenden AWS PrivateLink , ohne das öffentliche Internet zu durchqueren. MicroVMS unterstützt je nach gewünschtem Verkehrsziel zwei VPC-Endpunkte:

- MicroVM-Management-APIs (Images erstellen, ausführen, aussetzen, beenden) — verwendet den vorhandenen Lambda-VPC-Endpunkt (). `com.amazonaws.region.lambda`
- Konnektivität zu MicroVMS (HTTPS-Verkehr zu Ihren laufenden Anwendungen) — verwendet einen separaten Endpunkt (). `com.amazonaws.region.lambda-microvm`

VPC-Endpunkt für MicroVM-Management-APIs

Lambda MicroVMS nutzt denselben VPC-Endpunktdienst wie Lambda (). `com.amazonaws.region.lambda` Vollständige Anweisungen finden Sie unter Erstellen eines Schnittstellenendpunkts für Lambda.

Endpunktrichtlinie für MicroVM-Management-APIs

Um zu kontrollieren, wer Ihren Schnittstellenendpunkt verwenden kann und welche Lambda MicroVMS-API-Aktionen sie ausführen können, fügen Sie eine Endpunktrichtlinie hinzu. Die Richtlinie gibt den Prinzipal an, der Aktionen ausführen kann, die Aktionen, die er ausführen kann, und die Ressourcen, auf die er einwirken kann. Lambda MicroVMS-Aktionen verwenden das `lambda:` IAM-Aktionspräfix.

Weitere Informationen finden Sie unter Steuerung des Zugriffs auf Services mit VPC-Endpunkten im Amazon-VPC-Benutzerhandbuch.

Die folgende Beispielrichtlinie ermöglicht es dem Benutzer, `MyUser` MicroVM-Images aufzulisten und über den Endpunkt abzurufen:

```
{
  "Statement": [
    {
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/MyUser"
      },
      "Effect": "Allow",
      "Action": [
        "lambda:ListMicrovmImages",
        "lambda:GetMicrovmImage"
      ],
      "Resource": "*"
    }
  ]
}
```

VPC-Endpunkt für MicroVM-Konnektivität

Um den HTTPS-Verkehr zu Ihren laufenden MicroVMs privat zu halten, erstellen Sie einen Schnittstellenendpunkt für den Dienst. `com.amazonaws.region.lambda-microvm` Dieser Endpunkt verarbeitet Verbindungen zu MicroVM-Endpunkt-URLs (z. B.). `abc123def456.lambda-microvm.us-east-1.on.aws`

Weitere Informationen zu den Eigenschaften von Schnittstellenendpunkten finden Sie im Leitfaden zu Schnittstellenendpunkten in der Amazon VPC-Dokumentation.

Erstellen des Endpunkts

So erstellen Sie einen Schnittstellen-Endpunkt für MicroVM-Konnektivität (Konsole)

1. Öffnen Sie die Seite Endpunkte der Amazon-VPC-Konsole.
2. Wählen Sie Endpunkt erstellen aus.
3. Stellen Sie sicher, dass für Service-Kategorie die Option AWS -Services ausgewählt ist.
4. Wählen Sie für Servicename `com.amazonaws.region.lambda-microvm` aus. Stellen Sie sicher, dass der Typ Schnittstelle ist.
5. Auswählen von VPC und Subnetzen
6. Um privates DNS für den Schnittstellenendpunkt zu aktivieren, aktivieren Sie das Kontrollkästchen DNS-Name aktivieren (empfohlen). Dadurch wird sichergestellt, dass Anfragen, die den Hostnamen des öffentlichen MicroVM-Endpunkts verwenden, automatisch zu Ihrem Schnittstellenendpunkt aufgelöst werden, ohne dass clientseitige Änderungen erforderlich sind.

7. Wählen Sie unter Security group (Sicherheitsgruppe) eine oder mehrere Sicherheitsgruppen aus. Die Sicherheitsgruppe muss ausgehenden TCP-Verkehr über Port 443 zu den Endpunkt-Netzwerkschnittstellen zulassen.
8. Wählen Sie Endpunkt erstellen aus.

Um die private DNS-Option zu verwenden, müssen Sie die `enableDnsSupport` Attribute `enableDnsHostnames` und für Ihre VPC festlegen. Weitere Informationen finden Sie unter Anzeigen und Aktualisieren der DNS-Unterstützung für Ihre VPC im Amazon-VPC-Benutzerhandbuch.

So erstellen Sie einen Schnittstellenendpunkt für MicroVM-Konnektivität (AWS CLI)

```
aws ec2 create-vpc-endpoint \
  --vpc-id vpc-ec43eb89 \
  --vpc-endpoint-type Interface \
  --service-name com.amazonaws.us-east-1.lambda-microvm \
  --subnet-id subnet-abababab \
  --security-group-id sg-1a2b3c4d \
  --private-dns-enabled
```

Gehen Sie wie folgt vor, um zu überprüfen, ob der Endpunkt verfügbar ist und ob privates DNS aktiv ist:

```
aws ec2 describe-vpc-endpoints \
  --vpc-endpoint-ids vpce-1a2b3c4d5e6f7g8h9 \
  --query 'VpcEndpoints[0].
{State:State,PrivateDns:PrivateDnsEnabled,Dns:DnsEntries[*].DnsName}'
```

Verhalten bei privatem DNS

Wenn privates DNS aktiviert ist — Der Endpunkt verwaltet die DNS-Auflösung `*.lambda-microvm.region.on.aws` innerhalb Ihrer VPC. Ihre vorhandenen MicroVM-Endpunkt-Hostnamen werden (zum Beispiel`abc123def456.lambda-microvm.us-east-1.on.aws`) in die privaten IP-Adressen der Endpunkt-Netzwerkschnittstellen aufgelöst. Es ist keine Änderung des Clients erforderlich.

Wenn privates DNS deaktiviert ist, generiert Amazon VPC einen endpunktspezifischen DNS-Namen für Ihren Endpunkt in der Form. `vpce-id-hash.lambda-microvm.region.vpce.amazonaws.com` Um den Datenverkehr über diesen Endpunkt

weiterzuleiten und gleichzeitig die richtige MicroVM zu erreichen, müssen Sie den ursprünglichen MicroVM-Hostnamen an zwei Stellen beibehalten:

- TLS Server Name Indication (SNI) — Der TLS-Handshake verwendet diesen Wert, um zu identifizieren, für welche MicroVM die Verbindung bestimmt ist.
- HTTP-Host-Header — Der Proxy verwendet diesen Wert, um die Anfrage an die richtige MicroVM weiterzuleiten.

Wenn einer der Werte auf den VPC-Endpunkt-Hostnamen statt auf den MicroVM-Hostnamen gesetzt ist, kann die Verbindung nicht zur richtigen MicroVM weitergeleitet werden.

Beispiel: Connect über einen endpunktspezifischen DNS-Namen her

Im folgenden Beispiel wird curl mit dem `--connect-to` Flag verwendet, um die TCP-Verbindung zu Ihrem VPC-Endpunkt umzuleiten, wobei der MicroVM-Hostname in der URL, im SNI und im Host-Header beibehalten wird:

```
ENDPOINT_HOST=abc123def456.lambda-microvm.us-east-1.on.aws
VPCE_HOST=vpce-0a1b2c3d4e5f67890-a1b2c3d4.lambda-microvm.us-east-1.vpce.amazonaws.com

curl --connect-to "$ENDPOINT_HOST:443:$VPCE_HOST:443" \
  -H "x-aws-proxy-auth: $MICROVM_AUTH_TOKEN" \
  -H "x-aws-proxy-port: 8080" \
  "https://$ENDPOINT_HOST/"
```

Das `--connect-to` Flag weist curl an, die TCP-Verbindung zur VPC-Endpunktadresse zu öffnen, während die URL, TLS SNI und der Host-Header auf Ihren MicroVM-Hostnamen eingestellt bleiben.

Weitere Informationen finden Sie unter Zugriff auf einen Service über einen Schnittstellenendpunkt im Benutzerhandbuch für Amazon VPC.

Endpunktrichtlinie für MicroVM-Konnektivität

Sie können eine Endpunktrichtlinie anhängen, um zu steuern, welche MicroVMs über den `lambda-microvm` VPC-Endpunkt erreichbar sind. Mithilfe einer Endpunktrichtlinie für den `lambda-microvm` Service können Sie Verbindungen zu bestimmten Konten oder Organisationen festlegen. Standardmäßig erlaubt der Endpunkt Verbindungen zu MicroVMs in jedem AWS Konto. (Hinweis: Der Client, der die Verbindung herstellt, muss immer noch über ein gültiges MicroVM-Authentifizierungstoken verfügen, um Zugriff zu erhalten.)

Standardmäßig verfügt ein VPC-Endpunkt über eine Vollzugriffsrichtlinie, die den gesamten Datenverkehr zulässt. Wenn Sie die Standardrichtlinie durch eine benutzerdefinierte Richtlinie ersetzen, bewertet Lambda MicroVMS diese Richtlinie anhand der `lambda:ConnectMicrovm` Aktion bei jeder Verbindung, die über den Endpunkt hergestellt wird. Wenn die Richtlinie keine Verbindung zu MicroVMs zulässt, wird die Verbindung mit der Antwort HTTP 403 Forbidden abgelehnt. Eine Richtlinie, die keinen Zugriff gewährt, `lambda:ConnectMicrovm` verweigert alle Verbindungen über den Endpunkt.

> **Note**
>
> Die `lambda:ConnectMicrovm` Aktion autorisiert eine Verbindung zu einem MicroVM-Endpunkt über den Schnittstellenendpunkt. Es handelt sich nicht um einen Lambda-API-Vorgang und kann nicht in identitäts- oder ressourcenbasierten IAM-Richtlinien verwendet werden — er ist nur in einer VPC-Endpunktrichtlinie gültig.

Principal und Ressource

Verbindungen zu einem MicroVM-Endpunkt werden mit einem MicroVM-Authentifizierungstoken und nicht mit Signature Version 4 authentifiziert. AWS Aus diesem Grund ist der Verbindung kein IAM-Prinzipal zugeordnet. Stattdessen bewertet Lambda MicroVMS die Endpunktrichtlinie mit einem anonymen Prinzipal. Das bedeutet Folgendes:

- `Principal` muss `"*"` sein. Eine Richtlinie, die einen bestimmten Prinzipal benennt, entspricht nichts und verweigert jede Verbindung.
- Bedingungsschlüssel, die von der Identität des Anforderers abhängen (wie `aws:PrincipalArn`aws:PrincipalOrgID, und`aws:userid`), werden nicht ausgefüllt und stimmen nicht überein.
- `Resource`muss es auch sein. `"*"` Lambda MicroVMS beschränkt die Bewertung von Endpunktrichtlinien nicht auf einzelne MicroVM-Ressourcen-ARNs. Um einzuschränken, welche MicroVMs der Endpunkt erreichen kann, verwenden Sie den Bedingungsschlüssel und nicht das `aws:ResourceAccount` Element. `Resource`

Unterstützte Bedingungsschlüssel

Bedingungsschlüssel	Description
`aws:ResourceAccount`	Das AWS Konto, dem die MicroVM gehört, mit der eine Verbindung hergestellt wird.
`aws:ResourceOrgID`	Die AWS Organisations-ID des Kontos, dem die MicroVM gehört.
`aws:SourceVpce`	Die ID des Schnittstellenendpunkts, über den die Verbindung geleitet wurde.
`aws:SourceVpc`	Die ID der VPC, von der die Verbindung stammt.
`aws:VpcSourceIp`	Die private IP-Adresse des Clients, der die Verbindung hergestellt hat.

Beispiel: Erlauben Sie nur Verbindungen zu MicroVMs in Ihrem eigenen Konto

Die folgende Endpunktrichtlinie erlaubt Verbindungen über den Endpunkt nur zu MicroVMs, die dem Konto 111122223333 gehören. Verbindungen zu MicroVMs, die einem anderen Konto gehören, werden verweigert.

```
{
  "Statement": [
    {
      "Principal": "*",
      "Effect": "Allow",
      "Action": "lambda:ConnectMicrovm",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceAccount": "111122223333"
        }
      }
    }
  ]
}
```

Sicherheit und Berechtigungen

In diesem Abschnitt werden die IAM-Rollen, Authentifizierungstoken und Zugriffskontrollen für AWS Lambda MicroVMs beschrieben.

IAM-Rollen

Rolle erstellen

Diese Rolle wird bei der Image-Erstellung verwendet. Zum Abrufen von Quellartefakten und zum Generieren von Protokollen sind die folgenden Berechtigungen erforderlich: `s3:GetObject` `logs:CreateLogGrouplogs:CreateLogStream`, und`logs:PutLogEvents`. Fügen Sie auch hinzu`ecr:GetAuthorizationToken`, wenn auf private ECR-Bilder verwiesen wird.

Die Build-Rolle ist optional. Wenn Sie keine Build-Rolle angeben, kann Lambda keine Build-Logs in diese CloudWatch schreiben.

Rolle „Ausführung“

Diese Rolle wird zur MicroVM-Laufzeit verwendet, um Zugriffsberechtigungen für das Schreiben von Protokollen in CloudWatch und die Verwendung anderer AWS Dienste bereitzustellen.

Die Ausführungsrolle ist optional. Wenn Sie keine Ausführungsrolle angeben, gibt Lambda keine Laufzeitprotokolle an andere Dienste aus CloudWatch und Ihre MicroVM kann nicht auf andere AWS Dienste zugreifen.

Lifecycle-Hooks werden unter der Rolle ausgeführt, die ihrer Phase zugeordnet ist. Build-time Hooks (`/ready`und`/validate`) werden unter der Build-Rolle ausgeführt. Runtime-Hooks (`/run`,`/resume`,`/suspend`, und`/terminate`) werden unter der Ausführungsrolle ausgeführt.

Vertrauensrichtlinien

Für beide Rollen ist eine Vertrauensrichtlinie erforderlich, die es dem Lambda-Serviceprinzipal `lambda.amazonaws.com` ermöglicht, `sts:AssumeRole` und `sts:TagSession` auszuführen.

Authentifizierungstoken

Token steuern den eingehenden Zugriff auf laufende MicroVMs. Sie haben einen Portbereich; jedes Token gibt an, auf welche Ports es Zugriff gewährt.

```
# Token granting access to all ports
```

```
aws lambda-microvms create-microvm-auth-token \
  --microvm-identifier <id> \
  --expiration-in-minutes 30 \
  --allowed-ports '[{"allPorts":{}}]'

# Token granting access to port 8080 only
aws lambda-microvms create-microvm-auth-token \
  --microvm-identifier <id> \
  --expiration-in-minutes 30 \
  --allowed-ports '[{"port":8080}]'
```

Shell-Zugriffstoken

Der Shell-Zugriff verwendet eine separate Token-API. Die Ziel-MicroVM muss mit dem `SHELL_INGRESS` Netzwerkanschluss (`arn:aws:lambda:us-east-1:aws:network-connector:aws-network-connector:SHELL_INGRESS`) ausgeführt worden sein. Wenn die MicroVM nicht mit diesem Connector gestartet wurde, `create-microvm-shell-auth-token` schlägt Folgendes fehl: `ValidationException`

```
aws lambda-microvms create-microvm-shell-auth-token \
  --microvm-identifier <id> \
  --expiration-in-minutes 30
```

Eine vollständige Anleitung zum Herstellen einer Verbindung zu einer MicroVM-Shell finden Sie unter. the section called “Shell-Zugriff”

Taggen von Ressourcen

Lambda MicroVM-Images unterstützen AWS Ressourcen-Tags für Organisation und Kostenzuweisung. Fügen Sie Tags hinzu, wenn Sie ein Image mithilfe des `--tags` Parameters oder explizit nach der Erstellung erstellen. Sie können Tags auch auflisten und löschen, wie unten gezeigt: Sie können auch in IAM-Richtlinienbedingungen auf Tags verweisen, um den Zugriff auf MicroVM-Ressourcen zu kontrollieren.

```
# Add tags when creating an image
aws lambda-microvms create-microvm-image \
  --name my-app-image \
  --code-artifact '{"uri":"s3://my-bucket/app.zip"}' \
  --base-image-arn arn:aws:lambda:us-east-1:aws:microvm-image:al2023-1 \
  --build-role-arn arn:aws:iam::123456789012:role/MicrovmBuildRole \
```

```
  --tags '{"Environment":"production","Team":"platform"}'

# List tags on a resource
aws lambda-microvms list-tags \
  --resource arn:aws:lambda:us-east-1:123456789012:microvm-image:my-app-image

# Add tags to an existing resource
aws lambda-microvms tag-resource \
  --resource arn:aws:lambda:us-east-1:123456789012:microvm-image:my-app-image \
  --tags '{"CostCenter":"12345","Project":"sandbox-platform"}'

# Remove tags by key
aws lambda-microvms untag-resource \
  --resource arn:aws:lambda:us-east-1:123456789012:microvm-image:my-app-image \
  --tag-keys '["CostCenter","Project"]'
```

Referenz zu IAM-Berechtigungen

Lambda MicroVMS definiert die folgenden IAM-Aktionen. Verwenden Sie diese in IAM-Richtlinien, um den Zugriff auf MicroVM-Operationen zu steuern.

Action	Description
`lambda:CreateMicrovmImage`	Erstellen Sie ein neues MicroVM-Image.
`lambda:UpdateMicrovmImage`	Aktualisieren Sie ein vorhandenes MicroVM-Image.
`lambda:DeleteMicrovmImage`	Löschen Sie ein MicroVM-Image.
`lambda:GetMicrovmImage`	Ruft MicroVM-Image-Details ab.
`lambda:ListMicrovmImages`	Listet die MicroVM-Images im Konto auf.
`lambda:RunMicrovm`	Führen Sie eine neue MicroVM aus.
`lambda:GetMicrovm`	Rufen Sie den Status und die Details der MicroVM ab.
`lambda:ListMicrovms`	Listet MicroVMs im Konto auf.
`lambda:SuspendMicrovm`	Unterbrechen Sie eine laufende MicroVM.
`lambda:ResumeMicrovm`	Setzen Sie eine angehaltene MicroVM fort.

Action	Description
`lambda:TerminateMicrovm`	Beenden Sie eine MicroVM.
`lambda:CreateMicrovmAuthToken`	Generieren Sie ein Authentifizierungstoken für eine MicroVM.
`lambda:CreateMicrovmShellAuthToken`	Generieren Sie ein Shell-Zugriffstoken.

ARN-Formate für Ressourcen

```
# MicroVM image
arn:aws:lambda:<region>:<account>:microvm-image:<image-name>
# Network connector
arn:aws:lambda:<region>:<account>:network-connector:<connector-id>
```

Beispiel: Least-privilege Richtlinie für einen Operator

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "lambda:ListMicrovms",
          "lambda:ListMicrovmImages"
        ],
        "Resource": "*"
      },
      {
        "Effect": "Allow",
        "Action": [
          "lambda:RunMicrovm",
          "lambda:GetMicrovm",
          "lambda:SuspendMicrovm",
          "lambda:ResumeMicrovm",
          "lambda:TerminateMicrovm",
          "lambda:CreateMicrovmAuthToken",
          "lambda:GetMicrovmImage"
```

```
        ],
        "Resource": "arn:aws:lambda:*:123456789012:microvm-image:*"
      }
    ]
}
```

Überwachen

Diese Seite behandelt das CloudWatch Logging (Build und Runtime) und das CloudTrail Auditing für AWS Lambda MicroVMs.

Protokollierung (Protokolle) CloudWatch

AWS Lambda MicroVMS streamt Protokolle CloudWatch in zwei Phasen an:

Protokolle erstellen

Lambda generiert Build-Protokolle während der Image-Erstellung, einschließlich der Ausgabe von `Dockerfile` Ausführung, Anwendungsstart und Hook-Aufrufen. Diese Protokolle werden mithilfe der Build-Rolle gestreamt und in die Standard-Protokollgruppe geschrieben. `/aws/lambda-microvms/<image-name>`

Laufzeitprotokolle

Lambda streamt die Anwendungen stdout und stderr von laufenden MicroVMs nach. CloudWatch Laufzeitprotokolle werden mithilfe der Ausführungsrolle gestreamt und in dieselbe Standardprotokollgruppe geschrieben, wobei der Protokollstream standardmäßig die MicroVM-ID verwendet.

> **Important**
>
> Ohne eine Ausführungsrolle werden die Anwendungen stdout und stderr zur Laufzeit nicht weitergeleitet. CloudWatch Um Laufzeitprotokolle zu empfangen, geben Sie einer Ausführungsrolle die Berechtigungen`logs:CreateLogGroup`, und `logs:CreateLogStream` ein. `logs:PutLogEvents`

Benutzerdefinierte Protokollziele

Sie können eine benutzerdefinierte Protokollgruppe und einen benutzerdefinierten Stream konfigurieren, indem Sie den `--logging` Parameter on `run-microvm` oder verwenden`create-microvm-image`:

```
--logging '{"cloudWatch":{"logGroup":"/my-app/microvms","logStream":"custom-stream"}}'
```

Wenn Sie ihn weglassen`logStream`, wird standardmäßig die MicroVM-ID verwendet.

Sie können die Protokollierung explizit deaktivieren, indem Sie die folgende Konfiguration übergeben:

```
--logging '{"disabled":{}}'
```

CloudTrail Protokollierung

Lambda MicroVM ist in einen Dienst integriert CloudTrail, der eine Aufzeichnung der von einem Benutzer, einer Rolle oder einem AWS Dienst ausgeführten Aktionen bereitstellt. CloudTrail erfasst alle API-Aufrufe für Lambda MicroVMs als Ereignisse. Zu den erfassten Aufrufen gehören Aufrufe von der Lambda MicroVMS-Konsole und Codeaufrufe an die Lambda MicroVMS-API-Operationen.

Wenn Sie einen Trail erstellen, können Sie die kontinuierliche Übertragung von CloudTrail Ereignissen an einen Amazon S3-Bucket aktivieren, einschließlich Ereignissen für Lambda microVMs. Wenn Sie keinen Trail konfigurieren, können Sie die neuesten Ereignisse trotzdem in der CloudTrail Konsole im Event-Verlauf einsehen.

Anhand der gesammelten Informationen von können Sie ermitteln CloudTrail, welche Anfrage an Lambda MicroVMS gestellt wurde, die IP-Adresse, von der aus die Anfrage gestellt wurde, wer die Anfrage gestellt hat, wann sie gestellt wurde, und weitere Details.

CloudTrail zeichnet zwei Arten von Ereignissen für Lambda microVMs auf: Verwaltungsereignisse (Vorgänge auf Steuerungsebene wie das Erstellen und Löschen von Bildern) und Datenereignisse (Vorgänge auf Datenebene wie das Ausführen und Beenden von microVMs). Verwaltungsereignisse werden standardmäßig protokolliert. Datenereignisse erfordern eine ausdrückliche Anmeldung.

Verwaltungsereignisse

Lambda MicroVMS-Verwaltungsereignisse sind Operationen auf Steuerungsebene, die standardmäßig angemeldet werden. CloudTrail Zu den Verwaltungsereignissen gehören das

Erstellen, Aktualisieren und Löschen von MicroVM-Images sowie das Auflisten und Beschreiben von Ressourcen.

Die folgenden Lambda-MicroVMS-Aktionen werden als Verwaltungsereignisse protokolliert:

- `CreateMicrovmImage`
- `DeleteMicrovmImage`
- `DeleteMicrovmImageVersion`
- `UpdateMicrovmImage`
- `UpdateMicrovmImageVersion`
- `ListMicrovmImages`
- `GetMicrovmImage`
- `ListMicrovmImageVersions`
- `GetMicrovmImageVersion`
- `ListMicrovmImageBuilds`
- `GetMicrovmImageBuild`
- `ListMicrovms`
- `GetMicrovm`
- `ListManagedMicrovmImages`

Datenereignisse

Datenereignisse liefern Informationen über die Ressourcenoperationen, die an oder innerhalb einer Ressource ausgeführt werden (z. B. das Ausführen oder Beenden einer MicroVM). Sie werden auch als Vorgänge auf Datenebene bezeichnet. Datenereignisse sind oft Aktivitäten mit hohem Volume. Standardmäßig werden CloudTrail keine Datenereignisse protokolliert. Sie müssen die Protokollierung von Datenereignissen für Lambda-MicroVMs explizit aktivieren.

Der CloudTrail Ressourcentyp für Lambda MicroVMS-Datenereignisse ist. `AWS::Lambda::MicrovmImage` Sie können die CloudTrail Konsole oder die AWS CLI verwenden, um einen Trail- oder Event-Datenspeicher zum Protokollieren von Datenereignissen zu konfigurieren.

Die folgenden Lambda-MicroVMS-Aktionen werden als Datenereignisse protokolliert:

- `RunMicrovm`
- `TerminateMicrovm`

- `SuspendMicrovm`
- `ResumeMicrovm`
- `CreateMicrovmAuthToken`
- `CreateMicrovmShellAuthToken`

Verwenden Sie den folgenden erweiterten Eventselektor, um Datenereignisse für alle MicroVM-Images zu protokollieren:

```
{
   "Name": "Log Lambda MicroVM data events",
   "FieldSelectors": [
     { "Field": "eventCategory", "Equals": ["Data"] },
     { "Field": "resources.type", "Equals": ["AWS::Lambda::MicrovmImage"] }
   ]
}
```

Datenereignisprotokollierung aktivieren

Um die Protokollierung von Datenereignissen mithilfe der AWS CLI zu aktivieren, führen Sie den `put-event-selectors` Befehl mit erweiterten Event-Selektoren aus:

```
aws cloudtrail put-event-selectors \
   --trail-name my-trail \
   --advanced-event-selectors '[{
     "Name": "Log Lambda MicroVM data events",
     "FieldSelectors": [
       { "Field": "eventCategory", "Equals": ["Data"] },
       { "Field": "resources.type", "Equals": ["AWS::Lambda::MicrovmImage"] }
     ]
   }]'
```

Grundlegendes zu CloudTrail Protokolleinträgen

Ein CloudTrail Trail liefert API-Aktivitätsdatensätze an einen von Ihnen angegebenen Amazon S3-Bucket. Jeder Protokolleintrag stellt eine einzelne API-Anfrage dar und enthält die Anruferidentität, den Zeitstempel, die Anforderungsparameter und die Antwort.

Das folgende Beispiel zeigt einen CloudTrail Protokolleintrag, der ein Datenereignis für die `RunMicrovm` Aktion veranschaulicht:

```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "A1B2C3D4E5F6G7EXAMPLE:mySessionName",
    "arn": "arn:aws:sts::111122223333:assumed-role/MyRole/mySessionName",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2026-04-20T18:26:44Z",
  "eventSource": "lambda.amazonaws.com",
  "eventName": "RunMicrovm",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "127.0.0.1",
  "requestParameters": {
    "microvmImageArn": "arn:aws:lambda:us-east-1:111122223333:microvm-image:my-image"
  },
  "responseElements": {
    "microvmId": "ai-12345678-abcd-1234-ef56-123456789abc",
    "microvmState": "PENDING",
    "endpoint": "12345678-abcd-1234-ef56-123456789abc.lambda-microvm.us-east-1.on.aws",
    "microvmImageArn": "arn:aws:lambda:us-east-1:111122223333:microvm-image:my-image",
    "microvmImageVersion": "1.0"
  },
  "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
  "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
  "readOnly": false,
  "resources": [
    {
      "accountId": "111122223333",
      "type": "AWS::Lambda::MicrovmImage",
      "ARN": "arn:aws:lambda:us-east-1:111122223333:microvm-image:my-image"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "111122223333",
  "eventCategory": "Data"
}
```

Fehlerbehebung

In diesem Abschnitt wird beschrieben, wie Sie häufig auftretende Probleme bei der Arbeit mit AWS Lambda MicroVMs debuggen und beheben.

Shell-Zugriff

Verwenden Sie den Shell-Zugriff, um zum Debuggen und zur Fehlerbehebung eine direkte Verbindung zu einer laufenden MicroVM herzustellen.

Sie können auf zwei Arten eine Verbindung zu einer MicroVM-Shell herstellen:

- Konsole — Wählen Sie Ihre MicroVM in der Lambda-Konsole aus und wählen Sie Connect.
- CLI — Generieren Sie ein Shell-Token mit `create-microvm-shell-auth-token` und verwenden Sie das Token dann, um eine Verbindung herzustellen.

Generieren Sie ein Shell-Token und stellen Sie dann eine Verbindung her:

```
aws lambda-microvms create-microvm-shell-auth-token \
  --microvm-identifier <id> --expiration-in-minutes 30
# In Console: select MicroVM -> Connect
# In shell: ctr task ls, then ctr task exec -t --exec-id shell <id> /bin/sh
```

Die MicroVM muss mit dem `SHELL_INGRESS` Netzwerkanschluss (`arn:aws:lambda:us-east-1:aws:network-connector:aws-network-connector:SHELL_INGRESS`) ausgeführt worden sein. Wenn die MicroVM nicht mit diesem Connector gestartet wurde, wird ein `create-microvm-shell-auth-token` zurückgegeben. `ValidationException`

Für andere Probleme:

- Überprüfen Sie das `terminationMessage` Feld in der `get-microvm` Antwort auf terminierte MicroVMs.
- Überprüfen Sie die CloudWatch Build-Protokolle auf Probleme bei der Image-Erstellung.
- Überprüfen Sie das `StateReason` Feld auf Netzwerkanschlüsse im `FAILED` Status.

Fehlerbehebung

Dieser Abschnitt enthält Lösungen für häufig auftretende Probleme bei der Arbeit mit Lambda microVMs.

Symptom	Mögliche Ursache und Lösung
Die Image-Erstellung schlägt fehl (`CREATION_FAILED`)	Überprüfen Sie die Build-Protokolle unter `/aws/lambda-microvms/<image-name>`. Überprüfen Sie die `Dockerfile` Syntax, die Amazon S3-Berechtigungen und die Verfügbarkeit des Basis-Images. Zur Reproduktion `docker build` lokal ausführen.
MicroVM steckt fest `PENDING`	Warten Sie und versuchen Sie es erneut. Falls dies der Fall ist, überprüfen Sie den Zustand des Dienstes. Stellen Sie sicher, dass Ihr Kontingent für Parallelität nicht ausgeschöpft ist.
Die Bewerbung reagiert nach dem Lebenslauf nicht	Implementieren Sie den `/resume` Lifecycle-Hook, um die Verbindungen wiederherzustellen und den Status zu überprüfen. Stellen Sie nach dem Fortsetzen sicher, dass Ihre App an Port 8080 (oder den konfigurierten Port) gebunden ist.
502 Fehlerhaftes Gateway vom Endpunkt	Die Anwendung ist abgestürzt oder hört nicht zu. Überprüfen Sie die Laufzeitprotokolle. Verifizieren `EXPOSE` und `CMD` rein`Dockerfile`. Bei der automatischen Wiederaufnahme konnte die MicroVM möglicherweise nicht wieder aufgenommen werden (überprüfen Sie den Status mithilfe `get-microvm` von).
429 Zu viele Anfragen	Die Anforderungsrate wurde überschritten. Versuchen Sie es erneut mit exponentiellem Backoff und Jitter.
Verbindungen werden unterbrochen	Das Leerlauf-Timeout wurde ausgelöst. Implementieren Sie ping/pong Keepalives. Oder verlängern Sie die `maxIdleDurationSeconds` Idle-Richtlinie.

Symptom	Mögliche Ursache und Lösung
Hohe Latenz am Endpunkt	Bandbreitensättigung. Prüfen Sie, ob der Datenverkehr die Bandbreitenkapazität für Ihre MicroVM-Größe überschreitet. Skalieren Sie auf eine größere Größe.
Das Authentifizierungstoken ist abgelaufen (403)	Token haben ein konfigurierbares Ablaufdatum. Generieren Sie ein neues Token, bevor das alte abläuft. Implementieren Sie die Token-Aktualisierungslogik in Ihrem Client.
VPC-Ausgang funktioniert nicht	Stellen Sie sicher, dass der Netzwerkanschluss in `ACTIVE` Ordnung ist. Überprüfen Sie, ob die Sicherheitsgruppenregeln ausgehenden Verkehr zulassen. Vergewissern Sie sich, dass Subnetze Routen zu Ihren Zielressourcen haben.

Häufige Fehler (Image-Erstellung)

Fehler	Ursache	Lösung
`S3_ACCESS_DENIED`	Der Build-Rolle fehlen die Berechtigungen zum Abrufen des Amazon S3-Artefakts.	Fügen Sie `s3:GetObject` die Berechtigung für Ihren Artefakt-Bucket hinzu.
`S3_NO_SUCH_KEY`	Der Artefaktschlüssel ist im Bucket nicht vorhanden.	Stellen Sie sicher, dass der Amazon S3-Pfad korrekt ist.
`S3_NO_SUCH_BUCKET`	Der Amazon S3-Bucket ist nicht vorhanden.	Überprüfen Sie den Bucket-Na men und bestätigen Sie, dass er erstellt wurde.
`S3_INVALID_OBJECT`	Artefakt in der Speicherklasse Glacier oder in der Speicherk lasse, auf die nicht direkt zugegriffen werden kann.	Verschiebe das Artefakt in die Speicherklasse Standard.

Fehler	Ursache	Lösung
`S3_CROSS_REGION_ACCESS_DENIED`	Das Artefakt befindet sich in einer anderen Region als das MicroVM-Image.	Stellen Sie sicher, dass sich Ihr Artefakt in derselben Region wie Ihr MicroVM-Image befindet.
`ARCHIVE_DOCKERFILE_NOT_FOUND`	Im Zip-Archiv fehlt ein `Dockerfile` im Stammverzeichnis.	Fügen Sie a `Dockerfile` zum Stammverzeichnis Ihres ZIP-Archivs hinzu.
`ARCHIVE_INVALID`	Die Archivdatei ist kein gültiges ZIP-Archiv oder beschädigt.	Re-create das ZIP-Archiv und erneut hochladen.
`CONTAINER_BUILD_FAILED`	Ungültige `Dockerfile` Anweisungen, fehlende Dateien oder Syntaxfehler.	Debuggen Sie Ihre `Dockerfile` lokal mit`docker build`.
`DISK_STORAGE_FULL`	MicroVM ging während des Builds der Speicherplatz aus.	Reduzieren Sie die Größe der Artefakte oder wenden Sie sich an den Support.
`INTERNAL_PLATFORM_ERROR`	Es ist ein interner Fehler aufgetreten.	Wiederholen Sie den Vorgang. Wenn das Problem weiterhin besteht, wenden Sie sich an den Support.

Fehlerbehebung beim Netzwerkanschluss

Fehlercode	Ursache	Lösung
`DisallowedByVpcEncryptionControl`	Die VPC verfügt über eine Verschlüsselungskontrollrichtlinie, die unverschlüsselte Netzwerkschnittstellen oder Datenverkehr verhindert. Lambda kann keine ENIs erstellen,	Fügen Sie Lambda zur Ausschlussliste der VPC-Verschlüsselungskontrolle hinzu. Wenn ein Ausschluss nicht möglich ist, verwenden Sie eine VPC oder ein

Fehlercode	Ursache	Lösung
	die die Verschlüsselungsanforderungen erfüllen.	Subnetz, auf das keine restriktiven Verschlüsselungskontrollen angewendet wurden.
`Ec2RequestLimitExceeded`	Lambda führt EC2-API-Aufrufe durch (z. B.,`DescribeSubnets`)`CreateNetworkInterface` , um die Konnektivität einzurichten. Zu viele gleichzeitige EC2-API-Aufrufe führen zu einer Drosselung.	Versuchen Sie den Vorgang nach einer kurzen Verzögerung erneut. Wenn der Vorgang andauert, reduzieren Sie die gleichzeitigen Netzwerkverbindungsvorgänge oder fordern Sie beim Support eine Erhöhung des EC2-API-Drosselungslimits an. AWS
`InsufficientRolePermissions`	Für die Betreiberrolle fehlen die erforderlichen EC2-Berechtigungen.	Stellen Sie sicher, dass die IAM-Rolle über die erforderlichen EC2-Netzwerkberechtigungen verfügt.
`InternalError`	Während der Verarbeitung der Network Connector-Anfrage ist im Lambda-Dienst ein unerwarteter Fehler aufgetreten.	Wiederholen Sie den Vorgang. Wenn der Fehler auch nach mehreren Versuchen weiterhin besteht, wenden Sie sich mit dem ARN des Netzwerkkonnektors und dem ungefähren Zeitstempel an den AWS Support.
`InvalidSecurityGroup`	Die Sicherheitsgruppen-ID ist nicht vorhanden, wurde gelöscht oder gehört nicht zu derselben VPC wie die angegebenen Subnetze.	Stellen Sie sicher, dass alle Sicherheitsgruppen-IDs existieren und zu derselben VPC gehören wie die Subnetze. `aws ec2 describe-security-groups --group-ids <sg-id>` Zur Validierung verwenden.

Fehlercode	Ursache	Lösung
`InvalidSubnet`	Die Subnetz-ID ist nicht vorhanden, wurde gelöscht oder gehört zu einer anderen VPC als erwartet.	Stellen Sie sicher, dass alle Subnetz-IDs existieren und zur richtigen VPC gehören. `aws ec2 describe-subnets --subnet-ids <subnet-id>` Zur Validierung verwenden.
`SubnetOutOfIPAddresses`	Der CIDR-Block des Subnetzes ist erschöpft — alle IPs sind ENIs, Instances und anderen Ressourcen zugewiesen, sodass Lambda keine Netzwerkschnittstelle erstellen kann.	Geben Sie IP-Adressen frei, indem Sie ungenutzte entfernen ENIs/instances, oder verwenden Sie ein anderes Subnetz mit verfügbarer Kapazität. Ziehen Sie größere Subnetze (z. B. /24 oder größer) für Netzwerkanschlüsse in Betracht.

Bewährte Methoden für Lambda-MicroVMs

Folgen Sie diesen Best Practices, um zuverlässige, kostengünstige und sichere Anwendungen mit MicroVMS zu erstellen. AWS Lambda

Entwurf der Anwendung

Entwerfen Sie Ihre Anwendung so, dass sie mit MicroVM-Snapshots und Lifecycle-Hooks gut funktioniert:

- Verwenden Sie Lifecycle-Hooks — Implementieren Sie sie `/run` für die Initialisierung nach der Ausführung, `/suspend` für eine reibungslose Bereinigung, für die Wiederherstellung der Verbindung und `/resume` für das Löschen von Daten. `/terminate`
- Wiederverwendung von Snapshots handhaben — Generieren Sie UUIDs, Geheimnisse und Zufallswerte im Hook, nicht während der Image-Erstellung. `/run` Verwenden Sie CSPRNGs für die gesamte Generierung von Zufallszahlen.
- Entwurf für die Wiederaufnahme — Validiert alle Netzwerkverbindungen und den zwischengespeicherten Status im Hook. `/resume` AWS SDK-Verbindungen werden in der Regel automatisch wiederhergestellt, benutzerdefinierte Verbindungen jedoch möglicherweise nicht.

Leistung

Optimieren Sie die Leistung von MicroVM mit den folgenden Methoden:

- Right-size Ihre MicroVM — Wählen Sie die kleinste MicroVM-Größe, die Ihren CPU-, Speicher- und Bandbreitenanforderungen entspricht. Die Bandbreite skaliert mit dem konfigurierten Speicher.
- Minimierung der Snapshot-Größe — Halten Sie den Platzbedarf Ihrer Anwendung gering, um die Lauf- und Wiederaufnahmezeiten zu reduzieren. Entfernen Sie während der Erstellung bestehende Abhängigkeiten und temporäre Dateien, bevor der Image-Snapshot erfasst wird.
- Geeignet suspend/resume für latenzempfindliche Workloads — Halten Sie häufig genutzte Umgebungen im angehaltenen Zustand, um die Ausführung schnell wieder aufzunehmen.

Sicherheit

Folgen Sie diesen Methoden, um Ihre MicroVM-Workloads zu sichern:

- Authentifizierungstoken rotieren — Generieren Sie kurzlebige Token (15—30 Minuten). Implementieren Sie vor Ablauf die Token-Aktualisierungslogik in Ihrem Client.
- Gültigkeitsbereich von Tokens auf bestimmte Ports — Verwenden Sie den `allowedPorts` Parameter, um den Gültigkeitsbereich der Token auf die Ports zu beschränken, die Ihre Anwendung verwendet.
- Verwenden Sie ausgehenden VPC-Datenverkehr für vertraulichen Datenverkehr — Leiten Sie den Datenverkehr über einen VPC-Ausgangsconnector und nicht über das öffentliche Internet an Datenbanken und interne APIs weiter.
- Wenden Sie IAM mit den geringsten Rechten an — Trennen Sie die Build-Rollen (Image-Erstellung) von den Ausführungsrollen (Runtime). Gewähren Sie die erforderlichen Mindestberechtigungen.

Kostenoptimierung

Senken Sie die Kosten mit den folgenden Strategien:

- Richtlinien für den Leerlauf konfigurieren — Legen Sie fest`maxIdleDurationSeconds`, dass MicroVMs im Leerlauf automatisch angehalten werden. Für gesperrte MicroVMs fallen keine Rechenkosten an.

- Höchstwert festlegen DurationInSeconds — Vermeiden Sie unkontrollierbare Kosten, indem Sie für jede MicroVM eine feste Terminierungszeit festlegen.
- Ungenutzte MicroVMs beenden — Lassen Sie MicroVMs nicht unbegrenzt laufen. Wird `suspendedDurationSeconds` zum automatischen Beenden nach längeren Leerlaufzeiten verwendet.
- Ressourcen taggen — Verwenden Sie Tags auf MicroVM-Images für die Kostenzuweisung und Berichterstattung im AWS Cost Explorer.
- Right-size Ihre MicroVMs — Beginnen Sie mit der kleinsten Basislinie, die Ihren Leistungsanforderungen entspricht. Bei Spitzenauslastung können Sie die Leistung auf das Vierfache des Ausgangswerts erhöhen, ohne die Bereitstellung erneut durchführen zu müssen.
- Bereinigen Sie ungenutzte Images — Löschen Sie MicroVM-Image-Versionen, die Sie nicht mehr benötigen, um die Speicherkosten zu senken.

Überwachen

Überwachen Sie Ihre MicroVMs effektiv:

- Überprüfen Sie StateReason — Wenn MicroVMs unerwartet beendet werden, enthält die `get-microvm` Antwort eine Angabe `stateReason` mit dem Grund für den unerwarteten Ausfall.

Integrationen

AWS Lambda MicroVMS bietet isolierte, auf Snapshots basierende Rechenumgebungen, die KI-Agenten-Frameworks und Entwicklungstools als Sandboxes verwenden können. Jede Integration folgt demselben Muster: Sie erstellen ein wiederverwendbares MicroVM-Image, und das Framework startet eine MicroVM pro Sitzung oder Job über die Lambda-API.

Die folgenden Integrationen sind verfügbar:

- the section called “Von Claude verwaltete Agenten”— Verwenden Sie Lambda MicroVMS als selbst gehostete Sandbox für die Claude Managed Agents von Anthropic. Der Agenten-Loop läuft auf der Infrastruktur von Anthropic; die MicroVM führt Ihre Tool-Aufrufe durch. Ein Webhook-gesteuerter Controller startet eine MicroVM pro Sitzung.
- the section called “Cursor Cloud-Agenten”— Verwenden Sie Lambda MicroVMS als selbst gehostete Maschinen für Cursor Cloud Agents. Ein geplanter Controller fragt nach ausstehenden Poolanfragen ab und startet für jede Worker-Sitzung eine MicroVM.

- the section called “OpenAI Agenten-API”— Verwenden Sie Lambda MicroVMS als selbst gehostete Umgebung für Agenten, die mit der OpenAI Agents API erstellt wurden. Ihre Anwendung oder ein Webhook-Launcher startet für jede Runde eine MicroVM, optional mit der Option „Anhalten“ und „Fortfahren“ zwischen den Runden.

Verwendung von Lambda MicroVMS als Sandbox für Claude Managed Agents

AWS Lambda MicroVMS ist ein Managed-Sandbox-Anbieter, der selbst gehostete Sandboxen für Claude Managed Agents bereitstellt. Dabei werden vertrauliche Dateien, Pakete und Dienste in der von Ihnen kontrollierten Infrastruktur gespeichert. Anthropic hostet den Agenten-Loop und das Claude-Modell, während die Lambda-MicroVM der Ort ist, an dem Ihre Tool-Aufrufe ausgeführt werden. Mit diesem Muster kontrollieren Sie die Ausführungsumgebung — was installiert ist, welcher Netzwerkzugriff verfügbar ist und auf welche Ressourcen der Agent zugreifen kann.

Jede MicroVM ist eine Firecracker-isolated virtuelle Maschine mit einem Snapshot-basierten Start in Sekundenbruchteilen. Sie läuft bis zu 8 Stunden und kann beendet werden, wenn die Sitzung endet. Sitzungen haben niemals denselben Status. Mit dem serverlosen Betriebsmodell erhalten Sie die Sicherheitsgrenzen einer virtuellen Maschine — Sie müssen keine Cluster verwalten, keine ungenutzten Kapazitäten, für die Sie bezahlen müssen. Sie können MicroVMs mit AWS Diensten orchestrieren, um die selbst gehostete Sandbox-Spezifikation wie unten beschrieben zu implementieren. Weitere Informationen finden Sie im Referenzbeispiel für Claude Managed Agents Self-Hosted Sandboxes on Lambda microVMS.

Funktionsweise

Eine Steuerungsebene startet eine MicroVM pro Claude-Sitzung:

1. Eine Sitzung erreicht den Status Running und Anthropic sendet einen `session.status_run_started` Webhook an einen Endpunkt in Ihrem Konto.
2. Eine Launcher-Lambda-Funktion überprüft die Webhook-Signatur und ruft dann auf. `RunMicroVM`
3. Ihr Code auf der MicroVM beansprucht die Sitzung, führt Tool-Aufrufe (Bash, Lesen, Schreiben, Bearbeiten, Glob, Grep) in `/workspace` aus und sendet die Ergebnisse zurück an Anthropic.
4. Die MicroVM wird angehalten oder beendet, wenn die Sitzung endet.

Ihr API-Schlüssel für die Organisation von Anthropic erreicht AWS niemals Compute. Der Launcher übergibt nur eine AWS Secrets Manager-Referenz auf Ihren Anthropic-Umgebungsschlüssel. Die Ausführungsrolle der MicroVM ermöglicht es dem Code, ihn zur Laufzeit zu lesen.

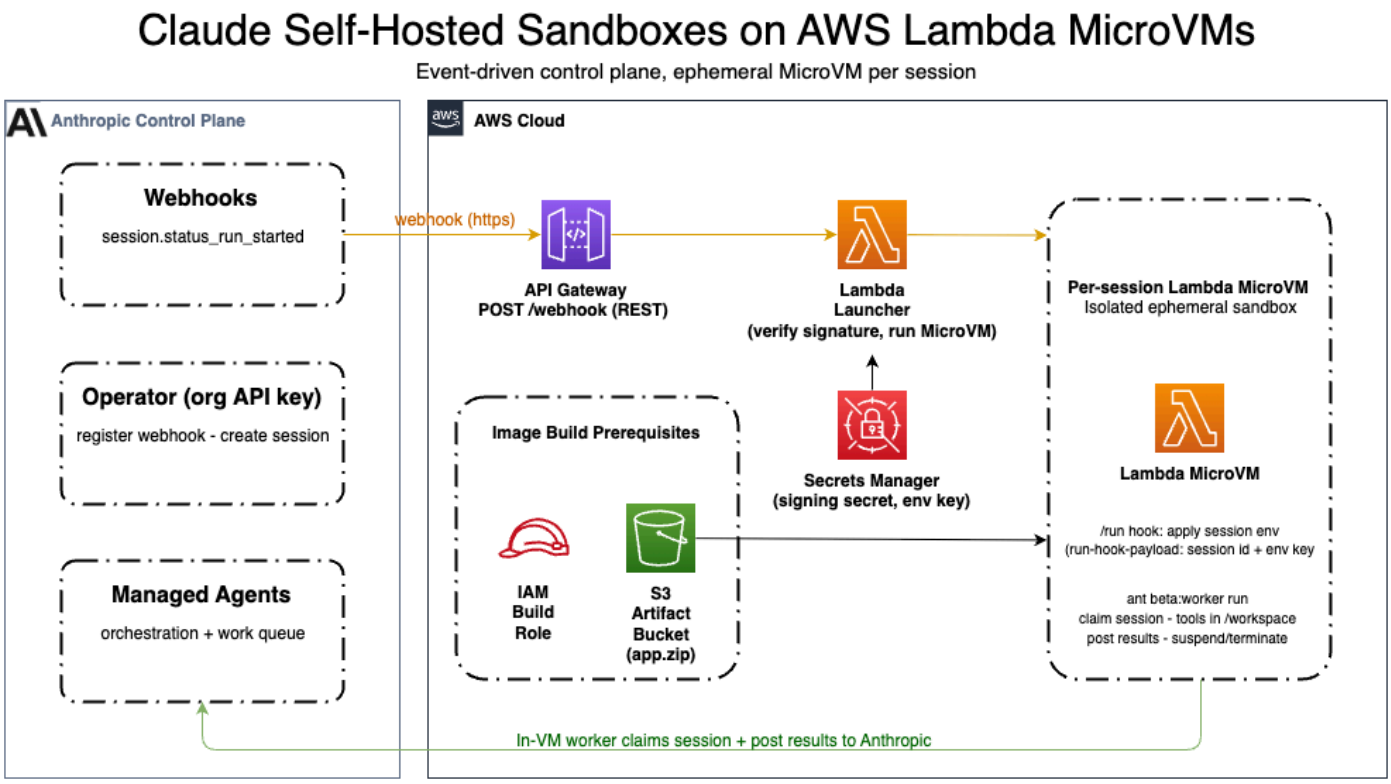
Die wichtigsten Eigenschaften

Eigenschaft	Vorteil
Isolierung von Feuerwerk skörpern	Hardware-virtualized Grenze pro Sitzung
Snapshot-based booten	Fahren Sie vom Firecracker-Snapshot in Sekundenbruchteilen bis einstelligen Sekunden fort
IAM über IMDSv2	Verwendet kurzfristige Anmeldeinformationen mit den geringsten Rechten
Dauer, die dem Status entspricht	Kann bis zu 8 Stunden bei vollem Festplatten- und Speicherz ugriff laufen
Pay-per-session	MicroVMs werden beendet, wenn die Sitzungen abgeschlossen sind. Dadurch endet die Abrechnung

Voraussetzungen

- Ein AWS Konto mit Berechtigungen für Amazon Simple Storage Service (Amazon S3), AWS Identity and Access Management (IAM), Secrets Manager, AWS WAF, Lambda und Lambda MicroVMS
- Ein Claude Managed Agents-Agent mit einer Umgebung `self_hosted`
- Ein Umgebungsschlüssel und ein geheimer Webhook-Signaturschlüssel aus der Claude-Konsole

Bereitstellung der Referenzimplementierung



Die [Claude Managed Agents Self-Hosted Sandboxes im Lambda] MicroVMS-Repository bieten eine minimale, funktionierende Bereitstellung. Sie umfasst:

- Ein CloudFormation Stack (Amazon API Gateway, Launcher-Lambda-Funktion, Secrets Manager-Geheimnisse, Amazon S3-Bucket, IAM-Rollen)
- Ein MicroVM-Image (`Dockerfile`, Node.js EnvironmentWorker Lifecycle-Hooks)
- Ein Bereitstellungsskript und ein Überprüfungsskript

Schritte zur Bereitstellung:

1. Stellen Sie den CloudFormation Stack bereit.
2. Speichern Sie den Umgebungsschlüssel und das Signaturgeheimnis in den erstellten Secrets Manager-Geheimnissen.
3. Erstellen Sie das MicroVM-Image.
4. Registrieren Sie die Webhook-URL des Stacks in der Claude-Konsole.
5. Überprüfen Sie dies, indem Sie eine Sitzung erstellen.

Eine ausführliche Anleitung finden Sie in der Readme-Datei des Repositorys.

Netzwerk

Lambda-MicroVMs verfügen standardmäßig über einen öffentlichen Internetzugang — es ist keine Konfiguration erforderlich, um dies zu erreichen. `api.anthropic.com`

Um auf private Ressourcen wie eine Amazon Aurora-Datenbank oder einen ElastiCache Amazon-Cluster zuzugreifen oder um Ihre eigenen Netzwerkeinschränkungen anzuwenden, fügen Sie beim Start einen VPC-Connector für ausgehenden Datenverkehr hinzu. Siehe Arbeiten mit ausgehenden Netzwerkverbindern.

Leerlaufrichtlinie

Stellen Sie `suspendedDurationSeconds: 0` und `autoResumeEnabled: false` für microVMs pro Sitzung ein. Wird `maximumDurationInSeconds` als Obergrenze für festsitzende Sitzungen festgelegt (max. 28.800 s).

```
{
  "maxIdleDurationSeconds": 120,
  "suspendedDurationSeconds": 0,
  "autoResumeEnabled": false
}
```

Überwachen

Anwendungsprotokolle — Überprüfen Sie die Anwendungsprotokolle in CloudWatch den Protokollen:

```
aws logs tail /aws/lambda-microvms/claude-worker --follow
```

MicroVMs ausführen — Listet die laufenden MicroVMs in Ihrem Konto auf:

```
aws lambda-microvms list-microvms \
  --image-identifier claude-worker \
  --query 'items[].[microvmId,state,startedAt]' --output table
```

Fehlerbehebung

Symptom	Ursache
Webhook gibt 401 zurück	Unterschreiben, geheime Diskrepanz oder veraltete Lieferung
MicroVM wird nicht gestartet	Webhook ist nicht registriert`session.status_run_started` , oder für die Rolle zur Ausführung der Lambda-Funktion für den Launcher fehlt die Berechtigung `RunMicroVM`
MicroVM wird sofort beendet	`/run`Hook-Timeout — erhöhen `runTimeoutInSeconds`
Der Arbeiter geht sofort	Ausgehendes HTTPS auf blockiert `api.anthropic.com`
Die Imageerstellung schlägt fehl `S3_*`	Fehlkonfiguration der Build-Rolle oder des Buckets

Verwenden von Lambda MicroVMS als Sandbox für Cursor Cloud Agents

Sie können AWS Lambda MicroVMs als selbst gehostete Maschinen für Ihre Cursor Cloud Agents verwenden, sodass die Repositorys, die Ausführung der Tools und der Netzwerkzugriff in der von Ihnen kontrollierten Infrastruktur verbleiben. Der Cursor hostet den Agenten-Loop und das Modell. Auf der Lambda-MicroVM beansprucht der Worker Poolanforderungen und führt Tool-Aufrufe aus. Mit diesem Muster steuern Sie die Ausführungsumgebung — was auf dem Image installiert ist, welcher Netzwerkzugriff verfügbar ist und welche AWS Ressourcen die IAM-Rolle des Workers erreichen kann.

Jede MicroVM ist eine Firecracker-isolated virtuelle Maschine mit einem Snapshot-basierten Start. Sie läuft bis zu 8 Stunden und wird beendet, wenn die Sitzung endet. Sitzungen haben niemals denselben Status. Mit dem serverlosen Betriebsmodell erhalten Sie die Sicherheitsgrenzen einer virtuellen Maschine — Sie müssen keine Cluster verwalten, keine ungenutzten Poolkapazitäten, für die Sie bezahlen müssen. Sie orchestrieren MicroVMs mit einer geplanten Controller-Lambda-Funktion, die auf ausstehende Poolanfragen reagiert. Weitere Informationen finden Sie in der Referenzvorlage „Self-hosted machines for Cursor Cloud Agents on AWS Lambda microVMS ausführen“.

Funktionsweise

Ein geplanter Controller startet eine MicroVM pro ausstehender Pool-Anfrage:

1. Sie starten einen Cloud-Agenten am Cursor. com/agentsgegen einen selbst gehosteten Computer. Die Anfrage bleibt ausstehend, bis sie von einem Mitarbeiter angefordert wird.
2. Die Lambda-Funktion des Controllers wird `agent worker controller --spawn ./spawn.sh` für ein fünfminütiges Abfragefenster für Server-Sent Ereignisse (SSE) ausgeführt. Wenn es eine passende Anfrage sieht, wird es ausgeführt. `spawn.sh`
3. `spawn.sh`ruft `RunMicrovm` (`aws lambda-microvms run-microvm`) auf und kehrt zurück, ohne zu warten. Es leitet die `CURSOR_*` Umgebung des Claims an den Gast weiter. `--run-hook-payload`
4. Der `/run` MicroVM-Hook (`hook.py`) wendet diese Nutzlast an und startet`entrypoint.sh`, was ausgeführt wird. `cursor-agent worker --pool … start` Der Worker führt Tool-Aufrufe in Ihrem Konto aus.
5. Wenn die Sitzung nach Ablauf des Release-Timeouts inaktiv ist, wird der Worker freigegeben und die MicroVM wird beendet.

Ihr API-Schlüssel für das Cursor-Dienstkonto wird im AWS Systems Manager Parameter Store als gespeichert. `SecureString` Der Controller und die MicroVM lesen ihn zur Laufzeit durch`ssm:GetParameter`; der Schlüssel wird nie in das Image integriert.

Die wichtigsten Eigenschaften

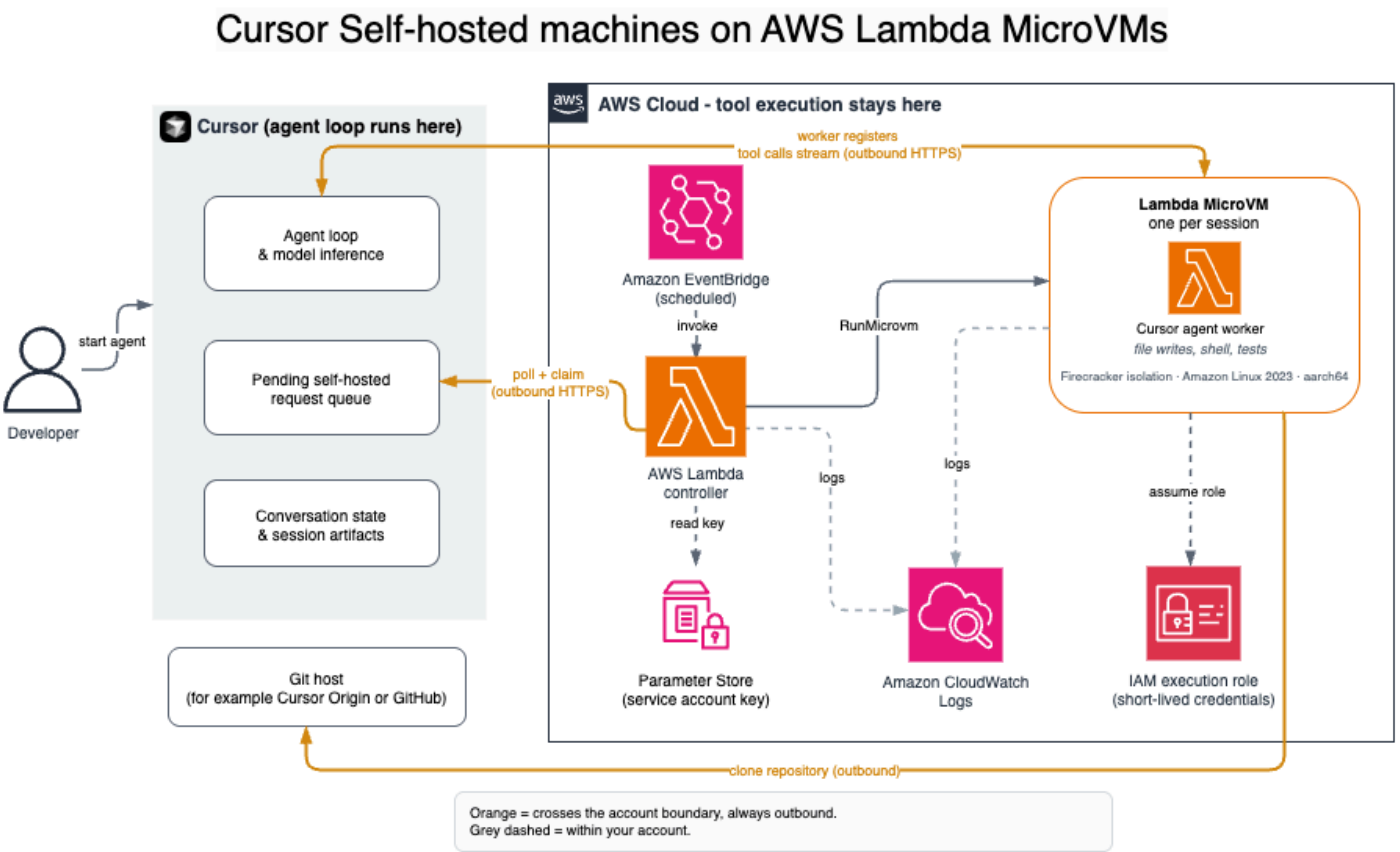
Eigenschaft	Vorteil
Isolierung von Feuerwerk skörpern	Hardware-virtualized Grenze pro Sitzung
Snapshot-Start	Lambda stellt den Gast aus einem Firecracker-Snapshot Ihres Images wieder her
IAM über die Ausführungsrolle	Der Gast verwendet kurzlebige Anmeldeinformationen von `MicroVmExecutionRoleArn`
Dauer des Zustands	Jede MicroVM kann bis zu 8 Stunden () laufen `--maximum-duration-in-seconds 28800`

Eigenschaft	Vorteil
Pay-per-session	Sie zahlen für die Laufzeit einer MicroVM, nicht für ungenutzte Poolkapazität

Voraussetzungen

- Ein AWS Konto mit aktivierten Lambda-MicroVMs sowie die Erlaubnis, Amazon S3, IAM und Systems Manager Parameter Store CloudFormation zu verwenden
- Ein Cursor Enterprise-Konto
- Ein Cursor-Team mit aktivierten selbst gehosteten Computern
- Ein Dienstkonto-API-Schlüssel für Pool-Mitarbeiter
- Die AWS CLI wurde auf die neueste stabile Version aktualisiert, und Docker (das Bereitstellungsskript erstellt das Controller-Image)

Bereitstellung der Referenzimplementierung



Das anysphere/aws -lambda-workers-Repository bietet eine minimale, funktionierende Bereitstellung. Sie umfasst:

- Ein CloudFormation Stack (`cloudformation.yaml`): eine geplante Controller-Lambda-Funktion, eine EventBridge `rate(1 minute)` Regel, ein Amazon S3-Artefakt-Bucket, eine Warteschlange mit toten Buchstaben und die IAM-Rollen (,,) `MicroVmExecutionRole` `BuildRole` `SpawnRole` `ControllerRole`
- Ein Controller-Image (`controller/Dockerfile`,`controller/handler.py`), das pro Aufruf ein SSE-Abfragefenster ausführt
- Ein MicroVM-Image (`microvm-image/`:`Dockerfile`,,`hook.py`) `entrypoint.sh`
- Ein Spawn-Hook (`spawn.sh`) und ein Deploy-Skript () `deploy.sh`

Schritte zur Bereitstellung:

1. Speichern Sie den API-Schlüssel für das Dienstkonto in SSM.

```
aws ssm put-parameter --type SecureString \
  --name /cursor-lambda-workers/cursor-api-key \
  --value "YOUR_SERVICE_ACCOUNT_KEY"
```

2. Stellen Sie den Stack bereit. `deploy.sh`erstellt das Controller-Image und wird ausgeführt`aws cloudformation deploy`.

```
POOL_NAMES=default ./deploy.sh
# POOL_NAMES=gpu,default ./deploy.sh
# ALL_POOLS=true ./deploy.sh
```

3. Erstellen Sie das MicroVM-Image **`microvm-image/`** mithilfe der Stack-Ausgaben. Aktivieren Sie die `ready` und `validate` Image-Hooks und den `run` MicroVM-Hook, damit sie im Snapshot-Pfad erfasst werden.

```
BUCKET=$(aws cloudformation describe-stacks --stack-name cursor-lambda-workers \
  --query "Stacks[0].Outputs[?OutputKey=='ArtifactBucketName'].OutputValue" --output
 text)
BUILD_ROLE=$(aws cloudformation describe-stacks --stack-name cursor-lambda-workers \
  --query "Stacks[0].Outputs[?OutputKey=='BuildRoleArn'].OutputValue" --output text)
BASE=$(aws lambda-microvms list-managed-microvm-images --query "items[0].imageArn" --
output text)
( cd microvm-image && zip -r /tmp/app.zip . )
```

```
aws s3 cp /tmp/app.zip "s3://${BUCKET}/app.zip"
aws lambda-microvms create-microvm-image \
  --code-artifact "uri=s3://${BUCKET}/app.zip" \
  --name cursor-pool-worker \
  --base-image-arn "${BASE}" \
  --build-role-arn "${BUILD_ROLE}" \
  --environment-variables "POOL_NAME=default,CURSOR_API_KEY_PARAM_NAME=/cursor-
lambda-workers/cursor-api-key" \
  --hooks '{"port":9000,"microvmImageHooks":
{"ready":"ENABLED","readyTimeoutInSeconds":60,"validate":"ENABLED","validateTimeoutInSeconds"
{"run":"ENABLED","runTimeoutInSeconds":60}}'
```

4. Starten Sie einen Cloud-Agenten vom Cursor aus. com/agentszur Überprüfung gegen den Pool (oder gegen ein Repository im repogebundenen Modus).

Eine ausführliche Anleitung finden Sie in der Readme-Datei des Repositorys.

Pool- und Repo-Modi

Achten Sie darauf, dass die Stacks `PoolNames` und die Images so `POOL_NAME` ausgerichtet sind, dass der Controller und der Gast denselben Pool bedienen.

- Any-repo Modus — Das Routing erfolgt nach Poolnamen, nicht nach Git Remote. Der Gast startet den Worker von einem Workspace ohne Git Remote aus, sodass der Worker `repo=` Labels weglässt. Benutzer wählen beim Starten eines Agenten die Gruppe und den Poolnamen Any Repo.
- Repo-bound Modus — Der Worker bedient eine oder mehrere bestimmte Git-Remotes. Entweder backen Sie den Klon in das MicroVM-Image ein oder legen Sie fest, `CURSOR_REPO_URL` dass er zu Beginn `entrypoint.sh` geklont wird. Der Worker leitet das `repo=` Label von Git Remote ab. Private Repositorys erfordern Git-Anmeldeinformationen für den Worker.

Netzwerk

Die Referenz verbindet `spawn.sh` den verwalteten `INTERNET_EGRESS` Connector, sodass der Worker die Steuerungsebene des Cursors erreichen kann, sowie `ALL_INGRESS` für die Image-Hooks. Outbound-only Worker erhalten nach dem Start niemals eingehendes HTTPS.

Um private Ressourcen (z. B. eine Amazon Aurora-Datenbank oder einen ElastiCache Amazon-Cluster) zu erreichen oder Ihre eigenen Einschränkungen anzuwenden, fügen Sie beim Start anstelle

von einen VPC-Connector für ausgehenden Datenverkehr hinzu. `INTERNET_EGRESS` Siehe the section called “Arbeiten mit ausgehenden Netzwerkverbindern”.

Überwachen

Gastprotokolle — Die MicroVM-Protokolle gehen zu den Protokollen: CloudWatch

```
aws logs tail /aws/lambda/microvms/cursor-pool-worker --follow
```

Controller-Protokolle — Die Controller-Lambda-Funktion protokolliert sich in. `/aws/lambda/cursor-lambda-workers-controller`

MicroVMS ausführen — Listet die laufenden MicroVMs für das Image auf:

```
aws lambda-microvms list-microvms --image-identifier cursor-pool-worker
```

Fehlerbehebung

Symptom	Ursache
Die Image-Erstellung schlägt fehl (S3 oder IAM)	Überprüfen Sie die Stack-Ausgaben `ArtifactBucketName` und`BuildRoleArn` ; die Zip-Datei muss in diesem Bucket landen und die Build-Rolle muss sie lesen können
MicroVM wird nicht gestartet	Der Controller läuft nicht oder kann nicht aufgerufen werden`RunMicrovm` . Stellen Sie sicher, dass das `cursor-pool-worker` Image existiert und (bei einem lokalen Controller) davon ausgegangen `SpawnRoleArn` wird
Worker beendet das Programm sofort	Der Gast fehlt `CURSOR_API_KEY` (SSM`/cursor-lambda-workers/cursor-api-key`), oder der `/run` Hook wurde nicht gestartet `cursor-agent worker … start`
`Exec format error` auf node	Das Image hat die falsche CLI-Architektur installiert; MicroVMs hier sind aarch64 () `arm64`
Service-account Der Schlüssel wurde als ungültig gemeldet	`CURSOR_API_ENDPOINT` /`CURSOR_API_URL` wurden an den Gast weitergeleitet; diese müssen nicht konfiguriert sein und `worker start` verwendet daher den Standard-Auth-Host

Symptom	Ursache
Das Image wurde erstellt, bevor Hooks aktiviert wurden	Erstellen Sie das MicroVM-Image neu `readyvalidate`, sodass sich die `run` Hooks im Snapshot-Pfad befinden

Zugehörige Ressourcen

- AWS Lambda-MicroVMs
- RunMicrovmCLI-Referenz
- Schnellstart für selbst gehostete Cursor-Maschinen
- Referenzvorlage: -lambda-workers anysphere/aws

Verwenden von Lambda MicroVMS als Sandbox für die OpenAI Agents API

Sie können AWS Lambda MicroVMS als selbst gehostete Umgebung für Agenten verwenden, die mit der OpenAI Agents API erstellt wurden. Workspace-Dateien, Tool-Ausführung und Netzwerkzugriff bleiben in der von Ihnen kontrollierten Infrastruktur erhalten. OpenAI führt den Agenten-Harness aus; die Lambda MicroVM führt Ihre Aufgaben aus. In diesem Beispiel wird es ausgeführt`codex exec-server`, das die Tool-Aufrufe des Agenten ausführt und die Workspace-Dateien der Sitzung enthält. Sie entscheiden über den Bildinhalt, die Netzwerkpfade, die der Executor verwenden kann, und über die AWS Ressourcen, auf die seine Ausführungsrolle beschränkt ist.

Jede MicroVM wird als Firecracker-isolated Umgebung ausgeführt, die von einem Snapshot Ihres Images aus gestartet wird. Die Gesamtlebensdauer einer MicroVM beträgt bis zu 8 Stunden, einschließlich Betrieb und Ruhezustand. Beenden Sie sie nach der letzten Runde oder setzen Sie sie zwischen den Runden aus. Sie erstellen ein wiederverwendbares Image und starten dann eine MicroVM entweder von Ihrer Anwendung oder einem Webhook-Launcher aus. Beide Ansätze verwenden dasselbe Image und dieselben Anmeldeinformationen. Weitere Informationen finden Sie im Referenzbeispiel für die selbst gehostete OpenAI Agents API-Sandbox auf Lambda MicroVMS.

Funktionsweise

Im vom Webhook verwalteten Pfad startet eine Launcher-Lambda-Funktion eine MicroVM, wenn die Sitzung zum ersten Mal einen Executor benötigt. Sie verwendet die vorhandene MicroVM wieder und nimmt sie zuerst wieder auf, wenn sie unterbrochen wird. Ihre Anwendung beendet es nach dem letzten Spielzug. Abbildung 1 zeigt den Ablauf:

1. Ihre Anwendung erstellt eine selbst gehostete Sitzung und sendet Eingaben an die Agenten-API.
2. Wenn die Sitzung einen Executor benötigt, sendet OpenAI einen `agent.session.action_required` Webhook mit einer `environment_connection` Aktion an einen Amazon API Gateway-Endpunkt in Ihrem Konto.
3. Eine Launcher-Lambda-Funktion überprüft die Webhook-Signatur, ruft die aktuelle Sitzung ab und bestätigt, dass sie dem Handler gehört.
4. Der Launcher ruft `RunMicrovm` mit dem Image, der Ausführungsrolle, den Netzwerkkonnektoren und einem, der `runHookPayload` die Verbindungswerte der Sitzung enthält, auf.
5. Der `/run` Hook der MicroVM verwendet seine IAM-Ausführungsrolle, um den Executor-Schlüssel aus AWS Secrets Manager abzurufen und Codex zu starten.
6. Der Executor registriert sich bei OpenAI unter. `https://api.openai.com` Anschließend öffnet er eine sichere WebSocket Verbindung, um Werkzeugbefehle `wss://codex-cloud-environments.chatgpt.com` zu empfangen und Ergebnisse zurückzugeben.
7. Ihre Anwendung kann Eingabedateien hochladen und Ausgabedateien über den authentifizierten HTTPS-Endpunkt der MicroVM herunterladen. Es folgt dem Sitzungsereignisstream und kann die MicroVM zwischen den Runden aussetzen.
8. Nach der letzten Runde beendet Ihre Anwendung die MicroVM.

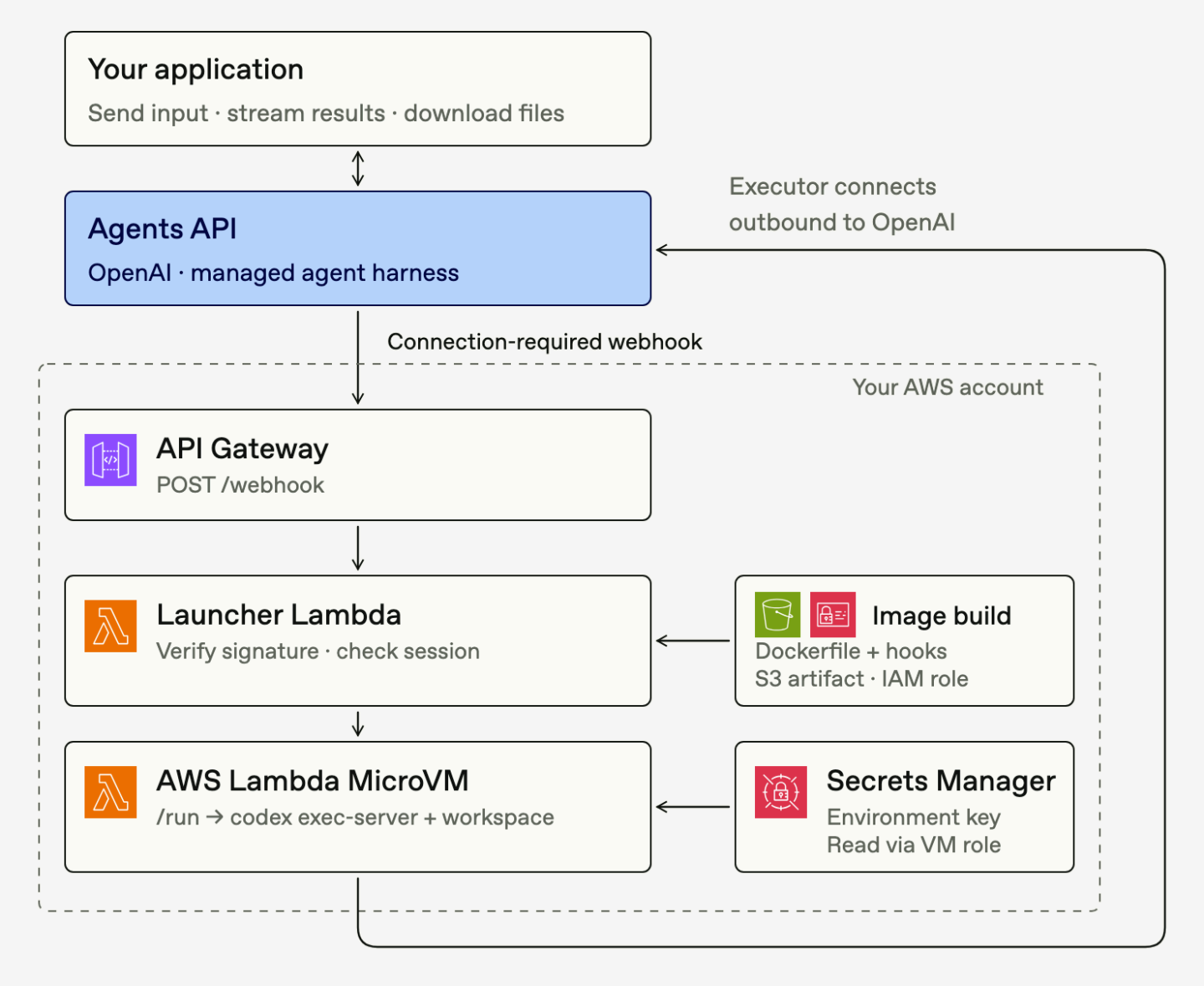


Abbildung 1. Webhook-managed Bereitstellung einer Agenten-API-Sandbox auf AWS Lambda MicroVMS.

Halten Sie den Anwendungsschlüssel und die Webhook-Signatur außerhalb der MicroVM geheim. Der Launcher übergibt nur den geheimen ARN des Executor-Schlüssels. Die VM ruft den Schlüssel mithilfe ihrer Ausführungsrolle ab.

Wichtige Eigenschaften

Eigenschaft	Vorteil
Isolierung von Feuerwerk skörpern	Hardware-virtualized Grenze für die MicroVM jeder Sitzung
Snapshot-based Start	Jede MicroVM wird von einem Snapshot Ihres vorbereiteten Images aus gestartet
IAM über die Ausführungsrolle	Der `/run` Hook liest den Executor-Schlüssel mit der Ausführun gsrolle der MicroVM; der Schlüssel wird nicht im Image gespeichert
Zustandsbehaftete Dauer	Jede MicroVM kann bis zu 8 Stunden laufen, einschließlich der Ruhezeit
Zwischen den Runden anhalten und fortsetzen	Arbeitsspeicher und Festplattenstatus bleiben erhalten, sodass nachfolgende Durchgänge in derselben Umgebung fortgesetzt werden
Usage-based berechnen	Für das Ausführen von VMs fallen Rechengebühren an. Durch die Sperrung fallen keine Rechenkosten mehr an; Snapshot-Speicher und Snapshot-Speicher reads/writes werden separat in Rechnung gestellt.

Voraussetzungen

- Ein AWS Konto mit Lambda-MicroVMS-Zugriff, eine AWS CLI-Version, die Folgendes beinhaltet`lambda-microvms`, und Berechtigungen zum Erstellen von Images und zum Ausführen von MicroVMs. Informationen zum Erstellen des Artefakt-Buckets von Amazon Simple Storage Service (Amazon S3) und der Image-Build-Rolle AWS Identity and Access Management (IAM) finden Sie unter. the section called “Erstellen Sie Ihre erste MicroVM”
- Ein OpenAI-Anwendungsschlüssel (`OPENAI_API_KEY`) zum Erstellen von Sitzungen und zum Senden von Eingaben.
- Ein separater OpenAI-Executor-Schlüssel (`OPENAI_EXECUTOR_API_KEY`) mit derselben Organisation, demselben Projekt und demselben Benutzer- oder Dienstkonto wie die Sitzung.

Speichern Sie es im Secrets Manager, zum Beispiel als. `codex/agents-api/executor` Der `/run` Hook übergibt es an Codex als`CODEX_API_KEY`.

- Eine MicroVM-Ausführungsrolle, die nur das geheime Executor-Schlüsselwort lesen kann. Wenn das Geheimnis einen vom Kunden verwalteten AWS Key Management Service (AWS AWS KMS) -Schlüssel verwendet, fügen Sie die Entschlüsselungsberechtigung hinzu. Siehe the section called “Sicherheit”.

Das MicroVM-Image wird vorbereitet

Erstellen Sie ein Image, das die Codex-CLI, Ihre Tool-Abhängigkeiten, ein Arbeitsverzeichnis wie `/workspace` und einen HTTP-Server für die Lifecycle-Hooks enthält.

Haken	Behavior
`/aws/lambda-microvms/runtime/v1/ready`	Gibt HTTP 200 zurück, wenn der Server bereit ist, dass Lambda den Snapshot erstellt. Der Executor bleibt getrennt.
`/aws/lambda-microvms/runtime/v1/run`	Liest die Verbindungswerte der Sitzung und den geheimen ARN, ruft den Executor-Schlüssel ab und startet den Executor.
`/aws/lambda-microvms/runtime/v1/validate`	Optional. Überprüft, ob Codex läuft und der Workspace beschreibbar ist.
`/aws/lambda-microvms/runtime/v1/suspend`	Erforderlich für. suspend/resume Stoppt den Executor und leert Schreibvorgänge, bevor AWS Snapshots der VM erstellt werden.
`/aws/lambda-microvms/runtime/v1/resume`	suspend/resumeErforderlich für. Ruft den Schlüssel erneut ab und startet den Executor für dieselbe Umgebung neu.
`/aws/lambda-microvms/runtime/v1/terminate`	Optional. Stoppt den Executor und leert Schreibvorgänge vor dem Beenden.

Verpacken Sie den `Dockerfile` und Hook-Server in ein Amazon S3-Artefakt und erstellen Sie dann das `codex-executor` Image mit 8 GB Basisspeicher (Basisspeicher mit 4 vCPUs). Aktivieren Sie die ausgewählten Hooks in der Image-Konfiguration und stellen Sie den Hook-Port so ein, dass er zu Ihrem Server passt. Halten Sie Sitzungs-IDs, Anmeldeinformationen und Live-Executor-Verbindungen vom Image-Snapshot fern. Informationen zur Build- und Hook-Konfiguration finden Sie unter. the section called “MicroVM-Images”

Beim Start serialisiert Ihre Anwendung oder Ihr Launcher die folgenden Werte als JSON in: `runHookPayload`

Wert	Zweck
`session.environment.id`	Identifiziert die Umgebung, mit der der Executor eine Verbindung herstellt
`session.environment.remote_url`	Stellt die OpenAI-Verbindungs-URL des Executors bereit
Executor-key geheimer ARN	Lässt den Hook den Schlüssel mithilfe der Ausführungsrolle der MicroVM abrufen

Kehren Sie vom `/run` Hook zurück, sobald der Executor-Prozess gestartet wurde, ohne auf das Ende der Runde zu warten, um einen Hook-Timeout zu vermeiden.

MicroVMS starten

Sie können MicroVMS von Ihrer Anwendung oder von einem Webhook-Launcher aus starten. Halten Sie in beiden Fällen den Event-Stream der Sitzung geöffnet, bis der Zug beendet ist.

Application-managed Bereitstellung

1. Erstellen Sie eine selbst gehostete Sitzung, deren Arbeitsverzeichnis dem Image entspricht, und öffnen Sie dann den zugehörigen Event-Stream.
2. Rufen Sie `RunMicrovm` den Image-ARN und die Version, die Ausführungsrolle, die Netzwerkkonnektoren und `runHookPayload` an. Speichern Sie den zurückgegebenen `microvmId` Wert zusammen mit der Sitzungs-ID.
3. Warte ab`agent.session.environment.connected`, sende dann die Eingabe und verfolge das Ergebnis des Zuges.

4. Rufe die Ausgabedateien ab und beende den Spielzug wie unter beschriebenthe section called “Beenden einer Runde”.

Webhook-managed Bereitstellung

Ihre Anwendung erstellt die Sitzung, öffnet ihren Stream und übermittelt Eingaben. Der Launcher erledigt den Rest, wenn der Webhook eintrifft.

1. Stellen Sie eine `POST /webhook` Route in Amazon API Gateway bereit, die den Launcher aufruft. Erteilen Sie dem Launcher die Berechtigung, seine Anmeldeinformationen zu lesen, MicroVMs vom ausgewählten Image aus zu überprüfen, zu starten, fortzusetzen und zu beenden, die MicroVM-Ausführungsrolle zu übergeben und die konfigurierten Netzwerkkonnektoren zu verwenden.
2. Registrieren Sie den Endpunkt für und. `agent.session.action_required` `agent.session.failed` Speichern Sie das Signaturgeheimnis des Endpunkts in Secrets Manager zusammen mit dem Anwendungsschlüssel des Launchers.
3. Vergleicht die Webhook-Signatur mit dem Rohtext der Anfrage, bevor ihr auf Sessions zugreift oder Compute startet. Rufen Sie dann die aktuelle Sitzung ab und bestätigen Sie, dass sie dem Handler gehört, indem Sie beispielsweise einen dedizierten gespeicherten Agenten zuordnen.
4. Falls immer `environment_connection` noch ausstehend, überprüfen Sie die aufgezeichnete MicroVM mit`GetMicrovm`. Setzen Sie den Vorgang fort, `ResumeMicrovm` wenn er unterbrochen ist. Rufen Sie `RunMicrovm` nur an, wenn keine VM aufgezeichnet ist, und speichern Sie dann ihre ID. Wenn das Speichern fehlschlägt, beenden Sie die VM, die Sie gestartet haben. Wenn der Executor vor dem Verbindungs-Timeout eine Verbindung herstellt, wird die Warteeingabe ohne erneutes Senden fortgesetzt.
5. Rufen Sie bei einem `agent.session.failed` Ereignis die aktuelle Sitzung ab und beenden Sie die aufgezeichnete MicroVM nur, wenn der Status „Fehlgeschlagen“ weiterhin lautet. Ignoriert gelöschte Sitzungen, gelöste Aktionen und Sitzungen, die einem anderen Handler gehören.

Das Referenzbeispiel speichert die MicroVM-ID in den Sitzungsmetadaten. Gleichzeitige Webhook-Lieferungen können zusätzliche VMs starten, bevor diese ID gespeichert wird. Vermeiden Sie vor dem Produktionseinsatz gleichzeitige Starts für dieselbe Sitzung und deduplizieren Sie Webhook-Ereignisse.

Beenden einer Runde

Abbildung 2 zeigt den MicroVM-Lebenszyklus für eine einzelne Runde und den optionalen Suspend- und Resume-Pfad für nachfolgende Turns.

Figure 2. MicroVM lifecycle for one Agents API turn

Single-turn path (solid) and multi-turn extension with suspend and resume (dashed)

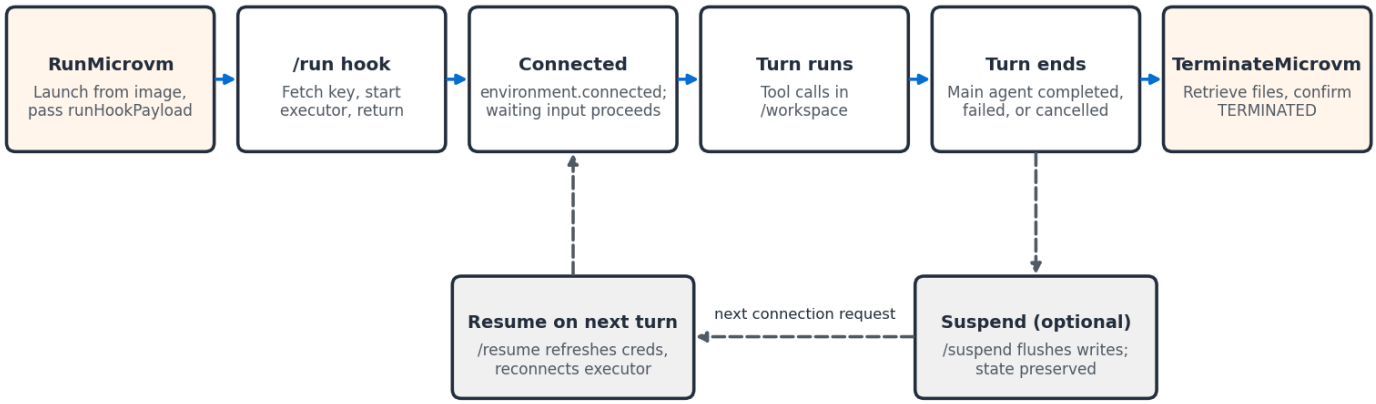


Abbildung 2. MicroVM-Lebenszyklus für eine Agenten-API-Runde. `RunMicrovm`startet die MicroVM, der `/run` Hook startet den Executor, die Umgebung stellt eine Verbindung her, der Turn läuft und der Turn endet. Im Single-Turn-Pfad rufen Sie Dateien ab und rufen auf. `TerminateMicrovm` Beim Multiturn-Pfad wird die MicroVM nach dem Turn unterbrochen und bei der nächsten Verbindungsanforderung wieder aufgenommen.

Warten Sie im Sitzungsstream auf oder auf den Hauptagenten `agent.session.turn.completedagent.session.turn.failed`, `agent.session.turn.cancelled` wo er sich befindet. `event.turn.subagent_id` `null` Unteragenten teilen sich die MicroVM, sodass ihre Terminalereignisse keine Säuberung auslösen. Rufen Sie die Dateien ab, die Sie benötigen. Rufen Sie nach dem letzten Zug an `TerminateMicrovm` und bestätigen Sie, dass die VM das Ziel erreicht`TERMINATED`. Lassen Sie die Maschine laufen oder setzen Sie sie aus, wenn Sie weitere Züge ausführen möchten. Führen Sie die Bereinigung durch, wenn in Ihrer Anwendung ein Fehler auftritt.

Turn-Ergebnisse kommen als Stream-Ereignisse an, nicht als Webhook-Abonnements. Der `agent.session.failed` Webhook deckt Sitzungsfehler ab, aber nicht jede fehlgeschlagene Runde. Aus diesem Grund sollten Sie den Vorgang nicht `agent.session.idle` alleine beenden, da er eintreffen kann, bevor die Warteeingabe beginnt.

Um die Lebensdauer der MicroVM zu begrenzen, falls die Bereinigung nicht ausgeführt wird, konfigurieren Sie beim Start die folgenden Einstellungen:

Einstellung	Empfehlung
`maximumDurationInSeconds`	Legen Sie eine Obergrenze für die Arbeitslast fest, z. B. 900 für einen 15-minütigen Test. Dieses Limit gilt auch während der laufenden Arbeit.
`maxIdleDurationSeconds`	Decken Sie die erwartete Arbeitsbelastung ab. Lambda misst den eingehenden Verkehr, sodass die ausgehende Verbindung des Executors diesen Timer nicht zurücksetzt.
`suspendedDurationSeconds` / `autoResumeEnabled`	Verwenden Sie 0/false für das Single-Turn-Muster, bei dem angehaltene MicroVMs nicht wieder aufgenommen werden.

Informationen zu Steuerungen über die gesamte Lebensdauer und den Leerlauf finden Sie unter. the section called “MicroVMS ausführen”

Löschen Sie die Sitzung separat. Durch das Löschen einer Sitzung wird die MicroVM nicht gestoppt und es wird kein Löschen-Webhook gesendet. Halten Sie das Image und den Executor-Schlüssel zur Wiederverwendung geheim. Informationen zum Umgang mit nachfolgenden Turns finden Sie unter Sandbox-Lebenszyklus.

Unterbrechen und Wiederaufnehmen zwischen den Zügen

Um den Speicher- und Festplattenstatus zwischen den Durchgängen beizubehalten, setzen Sie die MicroVM aus, nachdem der Haupt-Agent an der Reihe ist und alle Tools und Unteragenten fertig sind. Nehmen Sie dieselbe MicroVM wieder auf, wenn die nächste Sitzung einen Executor benötigt.

- Stellen Sie einen Wert ungleich Null ein. `suspendedDurationSeconds` Solange die MicroVM angehalten ist, fallen Snapshot-Speichergebühren an.
- Aktivieren Sie den `/suspend` Hook, um Schreibvorgänge zu löschen und Verbindungen zu schließen, und den `/resume` Hook, um Anmeldeinformationen zu aktualisieren und den Executor erneut zu verbinden.
- Der von der Anwendung verwaltete Pfad nimmt die virtuelle Maschine wieder auf, bevor die nächste Eingabe gesendet wird. Im vom Webhook verwalteten Pfad löst das Senden von Eingaben

die Verbindungsanforderung aus, mit der die VM wieder aufgenommen wird. Die Runde wird fortgesetzt, sobald der Executor die Verbindung wiederherstellt.

> **Note**
>
> `maximumDurationInSeconds`begrenzt die gesamte Laufzeit und die Dauer der Unterbrechung auf 8 Stunden. Um bei Bedarf weiterzumachen, rufen Sie an`ResumeMicrovm`. Für die automatische Wiederaufnahme ist eingehender Datenverkehr zur MicroVM erforderlich; die Agenten-API-Eingabe allein führt nicht zur Wiederaufnahme.

Netzwerk

Erlauben Sie dem Executor die erforderlichen ausgehenden Verbindungen zu OpenAI und den Zugriff auf Secrets Manager. Um private Ressourcen zu erreichen oder Ihre eigenen Netzwerkeinschränkungen anzuwenden, fügen Sie beim Start einen VPC-Anschluss für ausgehenden Datenverkehr hinzu. Siehe the section called “Arbeiten mit ausgehenden Netzwerkverbindern”.

Das Beispiel unterstützt das Hoch- und Herunterladen von Dateien über den HTTPS-Endpunkt der MicroVM, wobei ein Authentifizierungstoken verwendet wird, das auf den Port des Servers beschränkt ist.

Überwachen

Führen Sie eines der beiden Beispiele aus und bestätigen Sie dann, dass die MicroVM die Datei `hello.txt` herunterlädt und die gelöschte `TERMINATED` Sitzung HTTP 404 zurückgibt.

Launcher-Protokolle. Überprüfe die Logs der Launcher-Lambda-Funktion in CloudWatch Logs:

```
aws logs tail /aws/lambda/codex-agents-api-webhook --since 10m
```

MicroVMS ausführen. Stellen `MICROVM_IMAGE_ARN` Sie Ihr Image-ARN ein, verwenden Sie die AWS Region Ihrer Bereitstellung und listen Sie die MicroVMs für das Image auf:

```
aws lambda-microvms list-microvms \
  --image-identifier "$MICROVM_IMAGE_ARN" \
  --query 'items[].{id:microvmId,state:state}' --output table
```

Protokollieren Sie Sitzungs- und MicroVM-IDs zusammen, damit Sie einen Lauf von Ende zu Ende verfolgen können. Halten Sie Anmeldeinformationen und unformatierte Webhook-Texte aus den Protokollen fern.

Fehlerbehebung

Symptom	Was muss überprüft werden
Die Webhook-Signatur wird abgelehnt	Verwenden Sie das Signaturgeheimnis des Endpunkts und überprüfen Sie den unveränderten Anfragetext.
MicroVM wird nicht gestartet	Überprüfen Sie das Webhook-Abonnement, den Sitzungse igentumsfilter, die ausstehende `environment_connection` Aktion und die IAM-Berechtigungen des Launchers.
Das Erstellen des Images schlägt fehl	Überprüfen Sie das Amazon S3-Artefakt, die Image-Build-Rolle und die `/ready` Hook-Antwort.
`/run`schlägt fehl oder das Zeitlimit wird überschritten	Überprüfen Sie den geheimen Zugriff und den Start des Executors. Kehren Sie nach dem Start des Prozesses zurück, nicht nach dem Zug.
Executor kann keine Verbindung herstellen	Überprüfen Sie die Umgebungs-ID, die Remote-URL, den Besitz des Schlüssels und den ausgehenden Netzwerkzugriff.
MicroVM stoppt während einer Runde	Prüfen Sie die Richtlinien für maximale Lebensdauer und Leerlauf. Ausgehender Executor-Verkehr zählt nicht als eingehende Aktivität.

Prüfen Sie `agent.session.environment.failed` Ereignisse und die Executor-Protokolle auf Verbindungsfehler. Den Vertrag über den gemeinsamen Testamentsvollstrecker finden Sie unter Sandboxes. Self-hosted

Zugehörige Ressourcen

- MicroVMs
- the section called “MicroVMS ausführen”
- the section called “Sicherheit”

- RunMicrovm API-Referenz
- Selbst gehostete Sandboxen der OpenAI Agents API
- OpenAI Agents API im Handbuch AWS

Sicherheit bei AWS Lambda

Dieses Kapitel behandelt Sicherheit, Governance und Compliance für AWS Lambda. Die Richtlinien zu Datenschutz, Compliance, Resilienz und Infrastruktursicherheit gelten für alle Lambda-Compute-Typen. Zugriffskontrolle und Verschlüsselung haben rechnertypspezifische Details: Informationen zu Lambda Web Functions finden Sie unter the section called “Berechtigungen”the section called “Zugriffskontrolle”, undthe section called “Verschlüsselung im Ruhezustand”; für Lambda MicroVMS finden Sie unter. the section called “Sicherheit” In den verbleibenden Themen dieses Kapitels werden Lambda-Ereignisfunktionen beschrieben.

Cloud-Sicherheit AWS hat höchste Priorität. Als AWS Kunde profitieren Sie von einer Rechenzentrums- und Netzwerkarchitektur, die darauf ausgelegt sind, die Anforderungen der sicherheitssensibelsten Unternehmen zu erfüllen.

Sicherheit ist eine gemeinsame Verantwortung von Ihnen AWS und Ihnen. Das Modell der geteilten Verantwortung beschreibt dies als Sicherheit der Cloud und Sicherheit in der Cloud:

- Sicherheit der Cloud — AWS ist verantwortlich für den Schutz der Infrastruktur, die AWS-Services in der AWS Cloud läuft. AWS bietet Ihnen auch Dienste, die Sie sicher nutzen können. Third-party Auditoren testen und verifizieren regelmäßig die Wirksamkeit unserer Sicherheitsmaßnahmen im Rahmen der AWS Compliance-Programme. Weitere Informationen zu den Compliance-Programmen, die für gelten AWS Lambda, finden Sie AWS-Services unter Umfang nach Compliance-Programmen.
- Sicherheit in der Cloud: Ihr Verantwortungsumfang wird durch den AWS -Service bestimmt, den Sie verwenden. Sie sind auch für andere Faktoren verantwortlich, einschließlich der Vertraulichkeit Ihrer Daten, für die Anforderungen Ihres Unternehmens und für die geltenden Gesetze und Vorschriften.

Diese Dokumentation erläutert, wie das Modell der geteilten Verantwortung bei der Verwendung von Lambda zum Tragen kommt. Die folgenden Themen veranschaulichen, wie Sie Lambda zur Erfüllung Ihrer Sicherheits- und Compliance-Ziele konfigurieren können. Sie lernen auch, wie Sie andere verwenden können AWS-Services , die Ihnen helfen, Ihre Lambda-Ressourcen zu überwachen und zu sichern. Zu den Themen gehören Compliance-Validierung, Datenschutz, Identitäts- und Zugriffsmanagement, VPC-Schnittstellen-Endpunkte und Resilienz.

Weitere Informationen zur Anwendung von Sicherheitsprinzipien auf Lambda-Anwendungen finden Sie bei Serverless Land unter Sicherheit.

Topics

- Datenschutz in AWS Lambda
- Verwenden von serviceverknüpften Rollen für Lambda
- Identity and Access Management für AWS Lambda
- Erstellen Sie eine Governance-Strategie für Lambda-Funktionen und -Layer
- Konformitätsüberprüfung für AWS Lambda
- Resilienz in AWS Lambda
- Sicherheit der Infrastruktur in AWS Lambda
- Sicherung von Workloads mit öffentlichen Endpunkten
- Verwendung von Codesignatur zur Überprüfung der Codeintegrität mit Lambda

Datenschutz in AWS Lambda

> **Note**
>
> Die Verschlüsselungsdetails in diesem Thema, einschließlich der Verschlüsselung von Umgebungsvariablen, beschreiben Lambda-Ereignisfunktionen. Informationen zur Verschlüsselung von Lambda Web Functions finden Sie unterthe section called “Verschlüsselung im Ruhezustand”.

Für den Datenschutz in AWS Lambda gilt das Modell der geteilten Verantwortung das von AWS . Wie in diesem Modell beschrieben, AWS ist verantwortlich für den Schutz der globalen Infrastruktur, auf der die gesamte Infrastruktur ausgeführt wird. AWS Cloud Sie sind dafür verantwortlich, die Kontrolle über Ihre in dieser Infrastruktur gehosteten Inhalte zu behalten. Sie sind auch für die Sicherheitskonfiguration und die Verwaltungsaufgaben für die von Ihnen verwendeten AWS-Services verantwortlich. Weitere Informationen zum Datenschutz finden Sie unter Häufig gestellte Fragen zum Datenschutz . Weitere Informationen zum Datenschutz in Europa finden Sie im Zentrum für die Datenschutz-Grundverordnung (DSGVO).

Aus Datenschutzgründen empfehlen wir, dass Sie AWS-Konto Anmeldeinformationen schützen und einzelne Benutzer mit AWS IAM Identity Center oder AWS Identity and Access Management (IAM) einrichten. So erhält jeder Benutzer nur die Berechtigungen, die zum Durchführen seiner Aufgaben erforderlich sind. Außerdem empfehlen wir, die Daten mit folgenden Methoden schützen:

- Verwenden Sie für jedes Konto die Multi-Faktor-Authentifizierung (MFA).
- Wird verwendet SSL/TLS , um mit AWS Ressourcen zu kommunizieren. Wir benötigen TLS 1.2 und empfehlen TLS 1.3.
- Richten Sie die API und die Protokollierung von Benutzeraktivitäten mit ein AWS CloudTrail. Informationen zur Verwendung von CloudTrail Pfaden zur Erfassung von AWS Aktivitäten finden Sie unter Arbeiten mit CloudTrail Pfaden im AWS CloudTrail Benutzerhandbuch.
- Verwenden Sie AWS Verschlüsselungslösungen zusammen mit allen darin enthaltenen Standardsicherheitskontrollen AWS-Services.
- Verwenden Sie erweiterte verwaltete Sicherheitsservices wie Amazon Macie, die dabei helfen, in Amazon S3 gespeicherte persönliche Daten zu erkennen und zu schützen.
- Wenn Sie für den Zugriff AWS über eine Befehlszeilenschnittstelle oder eine API FIPS 140-3-validierte kryptografische Module benötigen, verwenden Sie einen FIPS-Endpunkt. Weitere Informationen über verfügbare FIPS-Endpunkte finden Sie unter Federal Information Processing Standard (FIPS) 140-3.

Wir empfehlen dringend, in Freitextfeldern, z. B. im Feld Name, keine vertraulichen oder sensiblen Informationen wie die E-Mail-Adressen Ihrer Kunden einzugeben. Dies gilt auch, wenn Sie mit Lambda oder anderen Geräten arbeiten und die Konsole, die API oder AWS SDKs AWS-Services verwenden. AWS CLI Alle Daten, die Sie in Tags oder Freitextfelder eingeben, die für Namen verwendet werden, können für Abrechnungs- oder Diagnoseprotokolle verwendet werden. Wenn Sie eine URL für einen externen Server bereitstellen, empfehlen wir dringend, keine Anmeldeinformationen zur Validierung Ihrer Anforderung an den betreffenden Server in die URL einzuschließen.

Abschnitte

- Verschlüsselung während der Übertragung
- Datenverschlüsselung im Ruhezustand für AWS Lambda

Verschlüsselung während der Übertragung

Lambda-API-Endpunkte unterstützen ausschließlich sichere Verbindungen über HTTPS. Wenn Sie Lambda-Ressourcen mit dem AWS-ManagementkonsoleAWS SDK oder der Lambda-API verwalten, wird die gesamte Kommunikation mit Transport Layer Security (TLS) verschlüsselt. Eine vollständige Liste der API-Endpunkte finden Sie unter AWS Regionen und Endpunkte in Allgemeine AWS-Referenz.

Wenn Sie Ihre Funktion mit einem Dateisystem verbinden, verwendet Lambda Verschlüsselung während der Übertragung für alle Verbindungen. Weitere Informationen finden Sie unter Datenverschlüsselung in Amazon EFS im Benutzerhandbuch zu Amazon Elastic File System.

Wenn Sie Umgebungsvariablen verwenden, können Sie Konsolenverschlüsselungshelfer aktivieren, um die clientseitige Verschlüsselung zu verwenden, um die Umgebungsvariablen während der Übertragung zu schützen. Weitere Informationen finden Sie unter Sicherung von Lambda-Umgebungsvariablen.

Datenverschlüsselung im Ruhezustand für AWS Lambda

AWS Lambda bietet immer Verschlüsselung im Ruhezustand für die folgenden Ressourcen:

- Umgebungsvariablen
- Dateien, die Sie zu Lambda hochladen, einschließlich Bereitstellungspakete und Ebenenarchive
- Objekte mit Filterkriterien für die Zuordnung von Ereignisquellen
- Dauerhafte Ausführungsdaten

> **Note**
>
> Für Funktionen und Ebenen, die selbstverwalteten S3-Codespeicher verwenden, speichert Lambda keine Kopie Ihres Quellcodes. Ihr Code verbleibt in Ihrem S3-Bucket, und die Verschlüsselung Ihres Quellcodes im Ruhezustand wird von Ihrer S3-Bucket-Konfiguration verwaltet. Lambda wird verwendet`KMSKeyArn`, um die entpackte Version des Pakets zu verschlüsseln, das Lambda zum Aufrufen der Funktion verwendet. Weitere Informationen finden Sie unter Schützen von Daten mit serverseitiger Verschlüsselung im Amazon Simple Storage Service-Benutzerhandbuch.

Sie können Lambda optional so konfigurieren, dass es einen vom Kunden verwalteten Schlüssel verwendet, um Ihre Umgebungsvariablen, ZIP-Bereitstellungspakete, Filterkriterienobjekte und dauerhafte Ausführungsdaten zu verschlüsseln.

Amazon CloudWatch protokolliert und AWS X-Ray verschlüsselt Daten standardmäßig und kann für die Verwendung eines vom Kunden verwalteten Schlüssels konfiguriert werden. Einzelheiten finden Sie unter Verschlüsseln von Protokolldaten in CloudWatch Protokollen und Datenschutz unter. AWS X-Ray

Überwachen Ihrer Verschlüsselungsschlüssel für Lambda

Wenn Sie einen vom AWS KMS Kunden verwalteten Schlüssel mit Lambda verwenden AWS CloudTrail, können Sie Die folgenden Beispiele sind CloudTrail Ereignisse für `DecryptDescribeKey`, und `GenerateDataKey` Aufrufe von Lambda für den Zugriff auf Daten, die mit Ihrem vom Kunden verwalteten Schlüssel verschlüsselt wurden.

Spezifische CloudTrail Beispiele für langlebige Ausführungsdaten finden Sie unterÜberwachung von KMS-Schlüsseln im Hinblick auf dauerhafte Funktionen.

Decrypt

Wenn Sie einen vom AWS KMS Kunden verwalteten Schlüssel zur Verschlüsselung Ihres Filterkriterienobjekts verwendet haben, sendet Lambda in Ihrem Namen eine `Decrypt` Anfrage, wenn Sie versuchen, im Klartext darauf zuzugreifen (z. B. über einen `ListEventSourceMappings` Anruf). Das folgende Beispielereignis zeichnet den Vorgang `Decrypt` auf:


```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:05:46Z",
```

```
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```

DescribeKey

Wenn Sie einen vom AWS KMS Kunden verwalteten Schlüssel zur Verschlüsselung Ihres Filterkriterienobjekts verwendet haben, sendet Lambda in Ihrem Namen eine `DescribeKey` Anfrage, wenn Sie versuchen, darauf zuzugreifen (z. B. über einen `GetEventSourceMapping` Anruf). Das folgende Beispielereignis zeichnet den Vorgang `DescribeKey` auf:

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2024-05-30T01:09:40Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "54.240.197.238",
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36",
    "requestParameters": {
        "keyId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
```

```
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "tlsDetails": {
        "tlsVersion": "TLSv1.3",
        "cipherSuite": "TLS_AES_256_GCM_SHA384",
        "clientProvidedHostHeader": "kms.eu-west-1.amazonaws.com"
    },
    "sessionCredentialFromConsole": "true"
}
```

GenerateDataKey

Wenn Sie einen vom AWS KMS Kunden verwalteten Schlüssel verwenden, um Ihr Filterkriterienobjekt in einem `CreateEventSourceMapping` `UpdateEventSourceMapping` OR-Anruf zu verschlüsseln, sendet Lambda in Ihrem Namen eine `GenerateDataKey` Anfrage zur Generierung eines Datenschlüssels zur Verschlüsselung der Filterkriterien (Umschlagverschlüsselung). Das folgende Beispielereignis zeichnet den Vorgang `GenerateDataKey` auf:

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:06:07Z",
                "mfaAuthenticated": "false"
            }
        },
```

```
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:04:18Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "numberOfBytes": 32,
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management"
}
```

Verwenden von serviceverknüpften Rollen für Lambda

Lambda verwendet serviceverknüpfte AWS Identity and Access Management Rollen (IAM). https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_terms-and-concepts.html#iam-term-service-linked-role Eine serviceverknüpfte Rolle ist eine einzigartige Art von IAM-Rolle, die direkt mit Lambda verknüpft ist. Service-linked Rollen werden von Lambda vordefiniert und beinhalten Berechtigungen, die der Dienst benötigt, um andere AWS Dienste in Ihrem Namen aufzurufen.

Lambda definiert die Berechtigungen seiner dienstverknüpften Rollen, und nur Lambda kann seine Rollen übernehmen. Die definierten Berechtigungen umfassen die Vertrauens- und Berechtigungsrichtlinie. Diese Berechtigungsrichtlinie kann keinen anderen IAM-Entitäten zugewiesen werden.

Sie können eine serviceverknüpfte Rolle erst löschen, nachdem ihre verwandten Ressourcen gelöscht wurden. Dies schützt Ihre Lambda-Ressourcen, da Sie nicht versehentlich die Zugriffsberechtigung für die Ressourcen entziehen können.

Informationen zu anderen Diensten, die serviceverknüpfte Rollen unterstützen, finden Sie unter AWS Dienste, die mit IAM funktionieren. Suchen Sie nach den Diensten, für die in der Rollenspalte Ja steht. Service-linked Wählen Sie über einen Link Ja aus, um die Dokumentation zu einer serviceverknüpften Rolle für diesen Service anzuzeigen.

Service-linked Rollenberechtigungen für Lambda

Lambda verwendet die dienstverknüpfte Rolle mit dem Namen. AWSServiceRoleForLambda Die serviceverknüpfte Rolle vertraut darauf, dass die folgenden Services die Rolle annehmen:

- `lambda.amazonaws.com`

Die benannte Richtlinie für Rollenberechtigungen AWSLambdaServiceRolePolicy ermöglicht es Lambda, die folgenden Aktionen für die angegebenen Ressourcen durchzuführen:

- Aktion: `ec2:TerminateInstances` aktiviert `arn:aws:ec2:*:*:instance/*` mit der Bedingung, die gleich ist `ec2:ManagedResourceOperator` `scaler.lambda.amazonaws.com`
- Aktion: `ec2:DescribeInstanceStatus` und `ec2:DescribeInstances` auf *

Sie müssen Berechtigungen konfigurieren, damit eine Benutzer, Gruppen oder Rollen eine serviceverknüpfte Rolle erstellen, bearbeiten oder löschen können. Weitere Informationen finden Sie im IAM-Benutzerhandbuch unter Service-linked Rollenberechtigungen.

Informationen zu verwalteten Richtlinienaktualisierungen finden Sie unter Von Lambda verwaltete Richtlinien.

Erstellen einer dienstverknüpften Rolle für Lambda

Sie müssen eine serviceverknüpfte Rolle nicht manuell erstellen. Wenn Sie einen Lambda-Kapazitätsanbieter in der AWS-Managementkonsole, der oder der AWS API erstellen AWS CLI, erstellt Lambda die serviceverknüpfte Rolle für Sie.

Wenn Sie diese serviceverknüpfte Rolle löschen und sie dann erneut erstellen müssen, können Sie dasselbe Verfahren anwenden, um die Rolle in Ihrem Konto neu anzulegen. Wenn Sie einen Lambda-Kapazitätsanbieter erstellen, erstellt Lambda die serviceverknüpfte Rolle erneut für Sie.

Sie können die IAM-Konsole auch verwenden, um eine serviceverknüpfte Rolle mit dem Anwendungsfall zu erstellen. AWSServiceRoleForLambda Erstellen Sie in der AWS CLI oder der AWS API eine serviceverknüpfte Rolle mit dem `lambda.amazonaws.com` Dienstnamen. Weitere Informationen finden Sie unter Erstellen einer serviceverknüpften Rolle im IAM-Benutzerhandbuch. Wenn Sie diese serviceverknüpfte Rolle löschen, können Sie mit demselben Verfahren die Rolle erneut erstellen.

Bearbeitung einer serviceverknüpften Rolle für Lambda

Lambda erlaubt Ihnen nicht, die serviceverknüpfte Rolle zu bearbeiten. AWSServiceRoleForLambda Da möglicherweise verschiedene Entitäten auf die Rolle verweisen, kann der Rollenname nach dem Erstellen einer serviceverknüpften Rolle nicht mehr geändert werden. Sie können jedoch die Beschreibung der Rolle mit IAM bearbeiten. Weitere Informationen finden Sie unter Bearbeiten einer serviceverknüpften Rolle im IAM-Benutzerhandbuch.

Löschen einer dienstverknüpften Rolle für Lambda

Wenn Sie ein Feature oder einen Service, die bzw. der eine serviceverknüpfte Rolle erfordert, nicht mehr benötigen, sollten Sie diese Rolle löschen. Auf diese Weise haben Sie keine ungenutzte Entität, die nicht aktiv überwacht oder verwaltet wird. Sie müssen jedoch die Ressourcen für Ihre serviceverknüpfte Rolle zunächst bereinigen, bevor Sie sie manuell löschen können.

> **Note**
>
> Wenn der Lambda-Service die Rolle verwendet, wenn Sie versuchen, die Ressourcen zu löschen, schlägt das Löschen möglicherweise fehl. Wenn dies passiert, warten Sie einige Minuten und versuchen Sie es erneut.

Um Lambda-Ressourcen zu löschen, die von der verwendet werden AWSServiceRoleForLambda

1. Entfernen Sie alle Lambda-Kapazitätsanbieter aus Ihrem Konto. Sie können dies mithilfe der Lambda-Konsole, CLI oder API tun.
2. Stellen Sie sicher, dass in Ihrem Konto keine Lambda-Kapazitätsanbieter mehr vorhanden sind, bevor Sie versuchen, die serviceverknüpfte Rolle zu löschen.

So löschen Sie die serviceverknüpfte Rolle mit IAM

Verwenden Sie die IAM-Konsole, die oder die AWS API AWS CLI, um die serviceverknüpfte Rolle zu löschen. AWSServiceRoleForLambda Weitere Informationen finden Sie unter Löschen einer serviceverknüpften Rolle im IAM-Benutzerhandbuch.

Unterstützte Regionen für serviceverknüpfte Lambda-Rollen

Lambda unterstützt nicht die Verwendung von serviceverknüpften Rollen in allen Regionen, in denen der Service verfügbar ist. AWSServiceRoleForLambda wird in den folgenden Regionen unterstützt.

Name der Region	Regions-ID	Unterstützung in Lambda
USA Ost (Nord-Virginia)	us-east-1	Ja
USA Ost (Ohio)	us-east-2	Ja
USA West (Nordkalifornien)	us-west-1	Ja
USA West (Oregon)	us-west-2	Ja
Afrika (Kapstadt)	af-south-1	Nein
Asien-Pazifik (Hongkong)	ap-east-1	Ja

Name der Region	Regions-ID	Unterstützung in Lambda
Asien-Pazifik (Jakarta)	ap-southeast-3	Ja
Asien-Pazifik (Bangkok)	ap-southeast-7	Ja
Asien-Pazifik (Mumbai)	ap-south-1	Ja
Asien-Pazifik (Osaka)	ap-northeast-3	Nein
Asien-Pazifik (Seoul)	ap-northeast-2	Nein
Asien-Pazifik (Singapur)	ap-southeast-1	Ja
Asien-Pazifik (Sydney)	ap-southeast-2	Ja
Asien-Pazifik (Tokyo)	ap-northeast-1	Ja
Kanada (Zentral)	ca-central-1	Nein
Europa (Frankfurt)	eu-central-1	Ja
Europa (Ireland)	eu-west-1	Ja
Europa (London)	eu-west-2	Ja
Europa (Milan)	eu-south-1	Nein
Europa (Paris)	eu-west-3	Nein
Europa (Stockholm)	eu-north-1	Nein
Naher Osten (Bahrain)	me-south-1	Nein
Naher Osten (VAE)	me-central-1	Nein
Südamerika (São Paulo)	sa-east-1	Nein
AWS GovCloud (US-East)	us-gov-east-1	Nein
AWS GovCloud (US-West)	us-gov-west-1	Nein

Identity and Access Management für AWS Lambda

> **Note**
>
> Die identitäts- und ressourcenbasierten Richtlinien in diesem Thema beschreiben Lambda Event Functions. Lambda-Webfunktionen verwenden ein anderes Zugriffskontrollmodell; siehe und. the section called “Berechtigungen” the section called “Zugriffskontrolle” Informationen zu Lambda MicroVMS finden Sie unter. the section called “Sicherheit”

AWS Identity and Access Management (IAM) hilft einem Administrator AWS-Service , den Zugriff auf Ressourcen sicher zu kontrollieren. AWS IAM-Administratoren steuern, wer authentifiziert (angemeldet) und autorisiert (Berechtigungen besitzt) ist, um Lambda-Ressourcen zu nutzen. IAM ist ein Programm AWS-Service , das Sie ohne zusätzliche Kosten nutzen können.

Themen

- Zielgruppe
- Authentifizierung mit Identitäten
- Verwalten des Zugriffs mit Richtlinien
- Wie AWS Lambda funktioniert mit IAM
- Identity-based politische Beispiele für AWS Lambda
- AWS verwaltete Richtlinien für AWS Lambda
- Fehlerbehebung AWS Lambda Identität und Zugriff

Zielgruppe

Wie Sie AWS Identity and Access Management (IAM) verwenden, hängt von Ihrer Rolle ab:

- Servicebenutzer – Fordern Sie von Ihrem Administrator Berechtigungen an, wenn Sie nicht auf Features zugreifen können (siehe Fehlerbehebung AWS Lambda Identität und Zugriff).
- Serviceadministrator – Bestimmen Sie den Benutzerzugriff und stellen Sie Berechtigungsanfragen (siehe Wie AWS Lambda funktioniert mit IAM).
- IAM-Administrator – Schreiben Sie Richtlinien zur Zugriffsverwaltung (siehe Identity-based politische Beispiele für AWS Lambda).

Authentifizierung mit Identitäten

Authentifizierung ist die Art und Weise, wie Sie sich AWS mit Ihren Identitätsdaten anmelden. Sie müssen sich als IAM-Benutzer authentifizieren oder eine IAM-Rolle annehmen. Root-Benutzer des AWS-Kontos

Sie können sich als föderierte Identität anmelden, indem Sie Anmeldeinformationen aus einer Identitätsquelle wie AWS IAM Identity Center (IAM Identity Center), Single Sign-On-Authentifizierung oder Anmeldeinformationen verwenden. Google/Facebook Weitere Informationen zum Anmelden finden Sie unter So melden Sie sich bei Ihrem AWS-Konto an im Benutzerhandbuch für AWS-Anmeldung .

AWS Bietet für den programmatischen Zugriff ein SDK und eine CLI zum kryptografischen Signieren von Anfragen. Weitere Informationen finden Sie unter AWS Signature Version 4 for API requests im IAM-Benutzerhandbuch.

AWS-Konto Root-Benutzer

Wenn Sie einen erstellen AWS-Konto, beginnen Sie mit einer Anmeldeidentität, dem sogenannten AWS-Konto Root-Benutzer, der vollständigen Zugriff auf alle AWS-Services Ressourcen hat. Wir raten ausdrücklich davon ab, den Root-Benutzer für Alltagsaufgaben zu verwenden. Eine Liste der Aufgaben, für die Sie sich als Root-Benutzer anmelden müssen, finden Sie unter Tasks that require root user credentials im IAM-Benutzerhandbuch.

Verbundidentität

Es hat sich bewährt, dass menschliche Benutzer für den Zugriff AWS-Services mithilfe temporärer Anmeldeinformationen einen Verbund mit einem Identitätsanbieter verwenden müssen.

Eine föderierte Identität ist ein Benutzer aus Ihrem Unternehmensverzeichnis, Ihrem Directory Service Web-Identitätsanbieter oder der AWS-Services mithilfe von Anmeldeinformationen aus einer Identitätsquelle zugreift. Verbundene Identitäten übernehmen Rollen, die temporäre Anmeldeinformationen bereitstellen.

Für die zentrale Zugriffsverwaltung empfehlen wir AWS IAM Identity Center. Weitere Informationen finden Sie unter Was ist IAM Identity Center? im AWS IAM Identity Center -Benutzerhandbuch.

IAM-Benutzer und -Gruppen

Ein IAM-Benutzer ist eine Identität mit bestimmten Berechtigungen für eine einzelne Person oder Anwendung. Wir empfehlen die Verwendung temporärer Anmeldeinformationen anstelle

von IAM-Benutzern mit langfristigen Anmeldeinformationen. Weitere Informationen finden Sie im IAM-Benutzerhandbuch unter Erfordern, dass menschliche Benutzer den Verbund mit einem Identitätsanbieter verwenden müssen, um AWS mithilfe temporärer Anmeldeinformationen darauf zugreifen zu können.

Eine IAM-Gruppe spezifiziert eine Sammlung von IAM-Benutzern und erleichtert die Verwaltung von Berechtigungen für große Gruppen von Benutzern. Weitere Informationen finden Sie unter Anwendungsfälle für IAM-Benutzer im IAM-Benutzerhandbuch.

IAM-Rollen

Eine IAM-Rolle ist eine Identität mit spezifischen Berechtigungen, die temporäre Anmeldeinformationen bereitstellt. Sie können eine Rolle übernehmen, indem Sie von einer Benutzer- zu einer IAM-Rolle (Konsole) wechseln AWS CLI oder einen AWS API-Vorgang aufrufen. Weitere Informationen finden Sie unter Methoden, um eine Rolle zu übernehmen im IAM-Benutzerhandbuch.

IAM-Rollen sind nützlich für den Verbundbenutzer-Zugriff, temporäre IAM-Benutzerberechtigungen, kontoübergreifenden Zugriff, serviceübergreifenden Zugriff und Anwendungen, die auf Amazon EC2 laufen. Weitere Informationen finden Sie unter Kontoübergreifender Ressourcenzugriff in IAM im IAM-Benutzerhandbuch.

Verwalten des Zugriffs mit Richtlinien

Sie kontrollieren den Zugriff, AWS indem Sie Richtlinien erstellen und diese an AWS Identitäten oder Ressourcen anhängen. Eine Richtlinie definiert Berechtigungen, wenn sie mit einer Identität oder Ressource verknüpft sind. AWS bewertet diese Richtlinien, wenn ein Principal eine Anfrage stellt. Die meisten Richtlinien werden AWS als JSON-Dokumente gespeichert. Weitere Informationen zu JSON-Richtliniendokumenten finden Sie unter Übersicht über JSON-Richtlinien im IAM-Benutzerhandbuch.

Mit Hilfe von Richtlinien legen Administratoren fest, wer Zugriff auf was hat, indem sie definieren, welches Prinzipal welche Aktionen auf welchen Ressourcenund unter welchen Bedingungendurchführen darf.

Standardmäßig haben Benutzer, Gruppen und Rollen keine Berechtigungen. Ein IAM-Administrator erstellt IAM-Richtlinien und fügt sie zu Rollen hinzu, die die Benutzer dann übernehmen können. IAM-Richtlinien definieren Berechtigungen unabhängig von der Methode, die zur Ausführung der Operation verwendet wird.

Identity-based Richtlinien

Identity-based Richtlinien sind Richtliniendokumente für JSON-Berechtigungen, die Sie an eine Identität (Benutzer, Gruppe oder Rolle) anhängen. Diese Richtlinien steuern, welche Aktionen Identitäten für welche Ressourcen und unter welchen Bedingungen ausführen können. Informationen zum Erstellen identitätsbasierter Richtlinien finden Sie unter Definieren benutzerdefinierter IAM-Berechtigungen mit vom Kunden verwalteten Richtlinien im IAM-Benutzerhandbuch.

Identity-based Richtlinien können Inline-Richtlinien (direkt in eine einzelne Identität eingebettet) oder verwaltete Richtlinien (eigenständige Richtlinien, die mehreren Identitäten zugeordnet sind) sein. Informationen dazu, wie Sie zwischen verwalteten und Inline-Richtlinien wählen, finden Sie unter Choose between managed policies and inline policies im IAM-Benutzerhandbuch.

Resource-based Richtlinien

Resource-based Richtlinien sind JSON-Richtliniendokumente, die Sie an eine Ressource anhängen. Beispiele hierfür sind Vertrauensrichtlinien für IAM-Rollen und Amazon S3-Bucket-Richtlinien. In Services, die ressourcenbasierte Richtlinien unterstützen, können Service-Administratoren sie verwenden, um den Zugriff auf eine bestimmte Ressource zu steuern. Sie müssen in einer ressourcenbasierten Richtlinie einen Prinzipal angeben.

Resource-based Richtlinien sind Inline-Richtlinien, die sich in diesem Dienst befinden. Sie können AWS verwaltete Richtlinien von IAM nicht in einer ressourcenbasierten Richtlinie verwenden.

Weitere Richtlinientypen

AWS unterstützt zusätzliche Richtlinientypen, mit denen die maximalen Berechtigungen festgelegt werden können, die durch gängigere Richtlinientypen gewährt werden:

- Berechtigungsgrenzen – Eine Berechtigungsgrenze legt die maximalen Berechtigungen fest, die eine identitätsbasierte Richtlinie einer IAM-Entität erteilen kann. Weitere Informationen finden Sie unter Berechtigungsgrenzen für IAM-Entitäten im -IAM-Benutzerhandbuch.
- Service-Kontrollrichtlinien (SCPs) – SCPs legen die maximalen Berechtigungen für eine Organisation oder Organisationseinheit in AWS Organizations fest. Weitere Informationen finden Sie unter Service-Kontrollrichtlinien im AWS Organizations -Benutzerhandbuch.
- Ressourcen-Kontrollrichtlinien (RCPs) – RCPs definieren die maximale Anzahl an Berechtigungen, die Ressourcen in Ihren Konten zur Verfügung stehen. Weitere Informationen finden Sie unter Ressourcen-Kontrollrichtlinien im AWS Organizations -Benutzerhandbuch.

- Sitzungsrichtlinien – Sitzungsrichtlinien sind erweiterte Richtlinien, die als Parameter übergeben werden, wenn Sie eine temporäre Sitzung für eine Rolle oder einen Verbundbenutzer erstellen. Weitere Informationen finden Sie unter Sitzungsrichtlinien im IAM-Benutzerhandbuch.

Mehrere Richtlinientypen

Wenn mehrere Arten von Richtlinien für eine Anfrage gelten, sind die daraus resultierenden Berechtigungen schwieriger zu verstehen. Informationen darüber, wie AWS bestimmt wird, ob eine Anfrage zulässig ist, wenn mehrere Richtlinientypen betroffen sind, finden Sie unter Bewertungslogik für Richtlinien im IAM-Benutzerhandbuch.

Wie AWS Lambda funktioniert mit IAM

Bevor Sie IAM zum Verwalten des Zugriffs auf Lambda verwenden, erfahren Sie, welche IAM-Features Sie mit Lambda verwenden können.

IAM-Feature	Lambda-Unterstützung
Identity-based Richtlinien	Ja
Resource-based Richtlinien	Ja
Richtlinienaktionen	Ja
Richtlinienressourcen	Ja
Richtlinienbedingungsschlüssel (services pezifisch)	Ja
ACLs	Nein
ABAC (Tags in Richtlinien)	Teilweise
Temporäre Anmeldeinformationen	Ja
Forward Access Sessions (FAS)	Nein
Servicerollen	Ja

IAM-Feature	Lambda-Unterstützung
Service-linked Rollen	Teilweise

Einen allgemeinen Überblick darüber, wie Lambda und andere AWS Dienste mit den meisten IAM-Funktionen funktionieren, finden Sie im AWS IAM-Benutzerhandbuch unter Dienste, die mit IAM funktionieren.

Identity-based Richtlinien für Lambda

Unterstützt Richtlinien auf Identitätsbasis: Ja

Identity-based Richtlinien sind Richtliniendokumente für JSON-Berechtigungen, die Sie an eine Identität anhängen können, z. B. an einen IAM-Benutzer, eine Benutzergruppe oder eine Rolle. Diese Richtlinien steuern, welche Aktionen die Benutzer und Rollen für welche Ressourcen und unter welchen Bedingungen ausführen können. Informationen zum Erstellen identitätsbasierter Richtlinien finden Sie unter Definieren benutzerdefinierter IAM-Berechtigungen mit vom Kunden verwalteten Richtlinien im IAM-Benutzerhandbuch.

Mit identitätsbasierten IAM-Richtlinien können Sie angeben, welche Aktionen und Ressourcen zugelassen oder abgelehnt werden. Darüber hinaus können Sie die Bedingungen festlegen, unter denen Aktionen zugelassen oder abgelehnt werden. Informationen zu sämtlichen Elementen, die Sie in einer JSON-Richtlinie verwenden, finden Sie in der IAM-Referenz für JSON-Richtlinienelemente im IAM-Benutzerhandbuch.

Identity-based Politikbeispiele für Lambda

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Identity-based politische Beispiele für AWS Lambda.

Resource-based Richtlinien innerhalb von Lambda

Unterstützt ressourcenbasierte Richtlinien: Ja

Resource-based Richtlinien sind JSON-Richtliniendokumente, die Sie an eine Ressource anhängen. Beispiele für ressourcenbasierte Richtlinien sind IAM-Rollen-Vertrauensrichtlinien und Amazon-S3-Bucket-Richtlinien. In Services, die ressourcenbasierte Richtlinien unterstützen, können Service-Administratoren sie verwenden, um den Zugriff auf eine bestimmte Ressource zu steuern. Für die Ressource, an welche die Richtlinie angehängt ist, legt die Richtlinie fest, welche Aktionen ein

bestimmter Prinzipal unter welchen Bedingungen für diese Ressource ausführen kann. Sie müssen in einer ressourcenbasierten Richtlinie einen Prinzipal angeben. Zu den Prinzipalen können Konten, Benutzer, Rollen, Verbundbenutzer oder gehören. AWS-Services

Um kontoübergreifenden Zugriff zu ermöglichen, können Sie ein gesamtes Konto oder IAM-Entitäten in einem anderen Konto als Prinzipal in einer ressourcenbasierten Richtlinie angeben. Weitere Informationen finden Sie unter Kontoübergreifender Ressourcenzugriff in IAM im IAM-Benutzerhandbuch.

Sie können eine ressourcenbasierte Richtlinie an eine Lambda-Funktion oder -Ebene anhängen. Diese Richtlinie legt fest, welche Prinzipale Aktionen auf der Funktion oder Ebene durchführen können.

Weitere Informationen zum Anfügen einer ressourcenbasierten Richtlinie an eine Funktion oder eine Ebene finden Sie unter Arbeiten mit ressourcenbasierten Richtlinien in Lambda.

Richtlinienaktionen für Lambda

Unterstützt Richtlinienaktionen: Ja

Administratoren können mithilfe von AWS JSON-Richtlinien angeben, wer Zugriff auf was hat. Das heißt, welcher Prinzipal Aktionen für welche Ressourcen und unter welchen Bedingungen ausführen kann.

Das Element `Action` einer JSON-Richtlinie beschreibt die Aktionen, mit denen Sie den Zugriff in einer Richtlinie zulassen oder verweigern können. Nehmen Sie Aktionen in eine Richtlinie auf, um Berechtigungen zur Ausführung des zugehörigen Vorgangs zu erteilen.

Eine Liste der Lambda-Aktionen finden Sie unter Von AWS Lambda definierte Aktionen in der Service-Autorisierungs-Referenz.

Richtlinienaktionen in Lambda verwenden das folgende Präfix vor der Aktion:

```
lambda
```

Um mehrere Aktionen in einer einzigen Anweisung anzugeben, trennen Sie sie mit Kommata:

```
"Action": [
      "lambda:action1",
      "lambda:action2"
```

```
        ]
```

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Identity-based politische Beispiele für AWS Lambda.

Richtlinienressourcen für Lambda

Unterstützt Richtlinienressourcen: Ja

Administratoren können mithilfe von AWS JSON-Richtlinien angeben, wer Zugriff auf was hat. Das heißt, welcher Prinzipal Aktionen für welche Ressourcen und unter welchen Bedingungen ausführen kann.

Das JSON-Richtlinienelement `Resource` gibt die Objekte an, auf welche die Aktion angewendet wird. Als Best Practice geben Sie eine Ressource mit dem zugehörigen Amazon-Ressourcennamen (ARN) an. Verwenden Sie für Aktionen, die keine Berechtigungen auf Ressourcenebene unterstützen, einen Platzhalter (*), um anzugeben, dass die Anweisung für alle Ressourcen gilt.

```
"Resource": "*"
```

Eine Liste der Lambda-Ressourcentypen und ihrer ARNs finden Sie unter Ressourcentypen definiert durch AWS Lambda in der Service Authorization Reference. Informationen zu den Aktionen, mit denen Sie den ARN einzelner Ressourcen angeben können, finden Sie unter Von AWS Lambda definierte Aktionen.

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Identity-based politische Beispiele für AWS Lambda.

Richtlinien-Bedingungsschlüssel für Lambda

Unterstützt servicespezifische Richtlinienbedingungsschlüssel: Ja

Administratoren können mithilfe von AWS JSON-Richtlinien angeben, wer auf was Zugriff hat. Das heißt, welcher Prinzipal Aktionen für welche Ressourcen und unter welchen Bedingungen ausführen kann.

Das Element `Condition` gibt an, wann Anweisungen auf der Grundlage definierter Kriterien ausgeführt werden. Sie können bedingte Ausdrücke erstellen, die Bedingungsoperatoren verwenden, z. B. ist gleich oder kleiner als, damit die Bedingung in der Richtlinie mit Werten in der

Anforderung übereinstimmt. Eine Übersicht aller AWS globalen Bedingungsschlüssel finden Sie unter Kontextschlüssel für AWS globale Bedingungen im IAM-Benutzerhandbuch.

Eine Liste von Lambda-Bedingungsschlüsseln finden Sie unter Bedingungsschlüssel für AWS Lambda in der Service-Autorisierungs-Referenz. Informationen zu den Aktionen und Ressourcen, mit denen Sie einen Bedingungsschlüssel verwenden können, finden Sie unter Aktionen definiert von AWS Lambda.

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Identity-based politische Beispiele für AWS Lambda.

ACLs in Lambda

Unterstützt ACLs: Nein

Zugriffssteuerungslisten (ACLs) steuern, welche Prinzipale (Kontomitglieder, Benutzer oder Rollen) auf eine Ressource zugreifen können. ACLs sind ähnlich wie ressourcenbasierte Richtlinien, verwenden jedoch nicht das JSON-Richtliniendokumentformat.

ABAC mit Lambda

Unterstützt ABAC (Tags in Richtlinien): Teilweise

Attribute-based Die Zugriffskontrolle (ABAC) ist eine Autorisierungsstrategie, die Berechtigungen auf der Grundlage von Attributen definiert, die als Tags bezeichnet werden. Sie können Tags an IAM-Entitäten und AWS -Ressourcen anhängen und dann ABAC-Richtlinien entwerfen, die Operationen zulassen, wenn das Tag des Prinzipals mit dem Tag auf der Ressource übereinstimmt.

Um den Zugriff auf der Grundlage von Tags zu steuern, geben Sie im Bedingungselement einer Richtlinie Tag-Informationen an, indem Sie die Schlüssel `aws:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, oder Bedingung `aws:TagKeys` verwenden.

Wenn ein Service alle drei Bedingungsschlüssel für jeden Ressourcentyp unterstützt, lautet der Wert für den Service Ja. Wenn ein Service alle drei Bedingungsschlüssel für nur einige Ressourcentypen unterstützt, lautet der Wert Teilweise.

Weitere Informationen zu ABAC finden Sie unter Definieren von Berechtigungen mit ABAC-Autorisierung im IAM-Benutzerhandbuch. Um ein Tutorial mit Schritten zur Einstellung von ABAC anzuzeigen, siehe Attributbasierte Zugriffskontrolle (ABAC) verwenden im IAM-Benutzerhandbuch.

Weitere Informationen über das Markieren von Lambda-Ressourcen mit Tags finden Sie unter Verwendung der attributbasierten Zugriffskontrolle in Lambda.

Verwenden temporärer Anmeldeinformationen mit Lambda

Unterstützt temporäre Anmeldeinformationen: Ja

Temporäre Anmeldeinformationen ermöglichen den kurzfristigen Zugriff auf AWS Ressourcen und werden automatisch erstellt, wenn Sie einen Verbund verwenden oder die Rollen wechseln. AWS empfiehlt, temporäre Anmeldeinformationen dynamisch zu generieren, anstatt langfristige Zugriffsschlüssel zu verwenden. Weitere Informationen finden Sie unter Temporäre Anmeldeinformationen in IAM und AWS-Services , die mit IAM funktionieren im IAM-Benutzerhandbuch.

Forward Access Sessions für Lambda

Unterstützt Forward Access Sessions (FAS): Nein

Forward Access Sessions (FAS) verwenden die Berechtigungen des Prinzipals, der einen aufruft AWS-Service, kombiniert mit der Anforderung, Anfragen AWS-Service an nachgeschaltete Dienste zu stellen. Einzelheiten zu den Richtlinien für FAS-Anforderungen finden Sie unter Zugriffssitzungen weiterleiten.

Servicerollen für Lambda

Unterstützt Servicerollen: Ja

Eine Servicerolle ist eine IAM-Rolle, die ein Service annimmt, um Aktionen in Ihrem Namen auszuführen. Ein IAM-Administrator kann eine Servicerolle innerhalb von IAM erstellen, ändern und löschen. Weitere Informationen finden Sie unter Erstellen einer Rolle zum Delegieren von Berechtigungen an einen AWS-Service im IAM-Benutzerhandbuch.

In Lambda wird eine Servicerolle als Ausführungsrolle bezeichnet.

> **Warning**
>
> Das Ändern der Berechtigungen für eine Ausführungsrolle kann die Lambda-Funktionalität beeinträchtigen.

Service-linked Rollen für Lambda

Unterstützt serviceverknüpfte Rollen: Ja

Eine dienstbezogene Rolle ist eine Art von Servicerolle, die mit einer AWS-Service verknüpft ist. Der Dienst kann die Rolle übernehmen, eine Aktion in Ihrem Namen auszuführen. Service-linked Rollen erscheinen in Ihrem Dienst AWS-Konto und gehören dem Dienst. Ein IAM-Administrator kann die Berechtigungen für Service-verknüpfte Rollen anzeigen, aber nicht bearbeiten.

Lambda verfügt im Gegensatz zu Lambda@Edge nicht über serviceverknüpfte Rollen. Weitere Informationen finden Sie unter Service-Linked Rollen für Lambda @Edge im Amazon CloudFront Developer Guide.

Details zum Erstellen oder Verwalten von serviceverknüpften Rollen finden Sie unter AWS -Services, die mit IAM funktionieren. Suchen Sie in der Tabelle nach einem Service, der einen `Yes` in der Service-linked Rollenspalte enthält. Wählen Sie den Link Yes (Ja) aus, um die Dokumentation für die serviceverknüpfte Rolle für diesen Service anzuzeigen.

Identity-based politische Beispiele für AWS Lambda

Standardmäßig verfügen Benutzer und Rollen nicht über die Berechtigung, Lambda-Ressourcen zu erstellen oder zu ändern. Ein IAM-Administrator muss IAM-Richtlinien erstellen, die Benutzern die Berechtigung erteilen, Aktionen für die Ressourcen auszuführen, die sie benötigen.

Informationen dazu, wie Sie unter Verwendung dieser beispielhaften JSON-Richtliniendokumente eine identitätsbasierte IAM-Richtlinie erstellen, finden Sie unter Erstellen von IAM-Richtlinien (Konsole) im IAM-Benutzerhandbuch.

Einzelheiten zu den von Lambda definierten Aktionen und Ressourcentypen, einschließlich des Formats der ARNs für die einzelnen Ressourcentypen, finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Lambda in der Service-Autorisierungs-Referenz.

Themen

- Best Practices für Richtlinien
- Verwenden von Lambda-Konsole
- Gewähren der Berechtigung zur Anzeige der eigenen Berechtigungen für Benutzer

Best Practices für Richtlinien

Identity-based Richtlinien legen fest, ob jemand Lambda-Ressourcen in Ihrem Konto erstellen, darauf zugreifen oder diese löschen kann. Dies kann zusätzliche Kosten für Ihr verursachen AWS-Konto.

Wenn Sie identitätsbasierte Richtlinien erstellen oder bearbeiten, befolgen Sie diese Richtlinien und Empfehlungen:

- Erste Schritte mit AWS verwalteten Richtlinien und Umstellung auf Berechtigungen mit den geringsten Rechten — Verwenden Sie die AWS verwalteten Richtlinien, die Berechtigungen für viele gängige Anwendungsfälle gewähren, um damit zu beginnen, Ihren Benutzern und Workloads Berechtigungen zu gewähren. Sie sind in Ihrem verfügbar. AWS-Konto Wir empfehlen Ihnen, die Berechtigungen weiter zu reduzieren, indem Sie vom AWS Kunden verwaltete Richtlinien definieren, die speziell auf Ihre Anwendungsfälle zugeschnitten sind. Weitere Informationen finden Sie unter Von AWS verwaltete Richtlinien oder Von AWS verwaltete Richtlinien für Auftragsfunktionen im IAM-Benutzerhandbuch.
- Anwendung von Berechtigungen mit den geringsten Rechten – Wenn Sie mit IAM-Richtlinien Berechtigungen festlegen, gewähren Sie nur die Berechtigungen, die für die Durchführung einer Aufgabe erforderlich sind. Sie tun dies, indem Sie die Aktionen definieren, die für bestimmte Ressourcen unter bestimmten Bedingungen durchgeführt werden können, auch bekannt als die geringsten Berechtigungen. Weitere Informationen zur Verwendung von IAM zum Anwenden von Berechtigungen finden Sie unter Richtlinien und Berechtigungen in IAM im IAM-Benutzerhandbuch.
- Verwenden von Bedingungen in IAM-Richtlinien zur weiteren Einschränkung des Zugriffs – Sie können Ihren Richtlinien eine Bedingung hinzufügen, um den Zugriff auf Aktionen und Ressourcen zu beschränken. Sie können beispielsweise eine Richtlinienbedingung schreiben, um festzulegen, dass alle Anforderungen mithilfe von SSL gesendet werden müssen. Sie können auch Bedingungen verwenden, um Zugriff auf Serviceaktionen zu gewähren, wenn diese für einen bestimmten Zweck verwendet werden AWS-Service, z. CloudFormation B. Weitere Informationen finden Sie unter IAM-JSON-Richtlinienelemente: Bedingung im IAM-Benutzerhandbuch.
- Verwenden von IAM Access Analyzer zur Validierung Ihrer IAM-Richtlinien, um sichere und funktionale Berechtigungen zu gewährleisten – IAM Access Analyzer validiert neue und vorhandene Richtlinien, damit die Richtlinien der IAM-Richtliniensprache (JSON) und den bewährten IAM-Methoden entsprechen. IAM Access Analyzer stellt mehr als 100 Richtlinienprüfungen und umsetzbare Empfehlungen zur Verfügung, damit Sie sichere und funktionale Richtlinien erstellen können. Weitere Informationen finden Sie unter Richtlinienvalidierung mit IAM Access Analyzer im IAM-Benutzerhandbuch.
- Multi-Faktor-Authentifizierung (MFA) erforderlich — Wenn Sie ein Szenario haben, das IAM-Benutzer oder einen Root-Benutzer in Ihrem System erfordert AWS-Konto, aktivieren Sie MFA für zusätzliche Sicherheit. Um MFA beim Aufrufen von API-Vorgängen anzufordern, fügen Sie Ihren

Richtlinien MFA-Bedingungen hinzu. Weitere Informationen finden Sie unter Sicherer API-Zugriff mit MFA im IAM-Benutzerhandbuch.

Weitere Informationen zu bewährten Methoden in IAM finden Sie unter Best Practices für die Sicherheit in IAM im IAM-Benutzerhandbuch.

Verwenden von Lambda-Konsole

Um auf die AWS Lambda Konsole zugreifen zu können, benötigen Sie ein Mindestmaß an Berechtigungen. Diese Berechtigungen müssen es Ihnen ermöglichen, Details zu den Lambda-Ressourcen in Ihrem AWS-Konto aufzulisten und anzuzeigen. Wenn Sie eine identitätsbasierte Richtlinie erstellen, die strenger ist als die mindestens erforderlichen Berechtigungen, funktioniert die Konsole nicht wie vorgesehen für Entitäten (Benutzer oder Rollen) mit dieser Richtlinie.

Sie müssen Benutzern, die nur die API AWS CLI oder die AWS API aufrufen, keine Mindestberechtigungen für die Konsole gewähren. Stattdessen sollten Sie nur Zugriff auf die Aktionen zulassen, die der API-Operation entsprechen, die die Benutzer ausführen möchten.

Eine Beispielrichtlinie, die minimalen Zugriff für die Funktionsentwicklung gewährt, finden Sie unter Benutzern Zugriff auf eine Event-Funktion gewähren. Zusätzlich zu den Lambda-APIs verwendet die Lambda-Konsole weitere Services, um Auslöserkonfigurationen anzuzeigen und neue Auslöser hinzuzufügen. Wenn Ihre Benutzer Lambda mit anderen Services nutzen, benötigen sie auch Zugriff auf diese Services. Details zum Konfigurieren weiterer Services mit Lambda finden Sie unter Lambda mit Ereignissen von anderen aufrufen AWS service.

Gewähren der Berechtigung zur Anzeige der eigenen Berechtigungen für Benutzer

In diesem Beispiel wird gezeigt, wie Sie eine Richtlinie erstellen, die IAM-Benutzern die Berechtigung zum Anzeigen der eingebundenen Richtlinien und verwalteten Richtlinien gewährt, die ihrer Benutzeridentität angefügt sind. Diese Richtlinie umfasst Berechtigungen zum Ausführen dieser Aktion auf der Konsole oder programmgesteuert mithilfe der API AWS CLI oder AWS .

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
```

```
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS verwaltete Richtlinien für AWS Lambda

Eine AWS verwaltete Richtlinie ist eine eigenständige Richtlinie, die von erstellt und verwaltet AWS wird. AWS Verwaltete Richtlinien dienen dazu, Berechtigungen für viele gängige Anwendungsfälle bereitzustellen, sodass Sie damit beginnen können, Benutzern, Gruppen und Rollen Berechtigungen zuzuweisen.

Beachten Sie, dass AWS verwaltete Richtlinien für Ihre speziellen Anwendungsfälle möglicherweise keine Berechtigungen mit den geringsten Rechten gewähren, da sie allen AWS Kunden zur Verfügung stehen. Wir empfehlen Ihnen, die Berechtigungen weiter zu reduzieren, indem Sie vom Kunden verwaltete Richtlinien definieren, die speziell auf Ihre Anwendungsfälle zugeschnitten sind.

Sie können die in AWS verwalteten Richtlinien definierten Berechtigungen nicht ändern. Wenn die in einer AWS verwalteten Richtlinie definierten Berechtigungen AWS aktualisiert werden, wirkt sich das

Update auf alle Prinzidentitäten (Benutzer, Gruppen und Rollen) aus, denen die Richtlinie zugeordnet ist. AWS aktualisiert eine AWS verwaltete Richtlinie höchstwahrscheinlich, wenn eine neue Richtlinie eingeführt AWS-Service wird oder neue API-Operationen für bestehende Dienste verfügbar werden.

Weitere Informationen finden Sie unter Von AWS verwaltete Richtlinien im IAM-Benutzerhandbuch.

Themen

- AWS verwaltete Richtlinie: AWS Lambda_FullAccess
- AWS verwaltete Richtlinie: AWS Lambda_ReadOnlyAccess
- AWS verwaltete Richtlinie: AWSLambdaBasicExecutionRole
- AWS verwaltete Richtlinie: AWSLambdaBasicDurableExecutionRolePolicy
- AWS verwaltete Richtlinie: AWSLambdaDynamoDBExecutionRole
- AWS verwaltete Richtlinie: AWSLambdaENIManagementAccess
- AWS verwaltete Richtlinie: AWSLambdaInvocation-DynamoDB
- AWS verwaltete Richtlinie: AWSLambdaKinesisExecutionRole
- AWS verwaltete Richtlinie: AWSLambdaMSKExecutionRole
- AWS verwaltete Richtlinie: AWSLambdaRole
- AWS verwaltete Richtlinie: AWSLambdaSQSQueueExecutionRole
- AWS verwaltete Richtlinie: AWSLambdaVPCAccessExecutionRole
- AWS verwaltete Richtlinie: AWSLambdaManagedEC2ResourceOperator
- AWS verwaltete Richtlinie: AWSLambdaServiceRolePolicy
- AWS verwaltete Richtlinie: AWSLambdaNetworkConnectorOperatorPolicy
- AWS verwaltete Richtlinie: AWSLambdaInvokeWebFunctionEndpointAccess
- Lambda-Updates für AWS Verwaltete Richtlinien

AWS verwaltete Richtlinie: AWS Lambda_FullAccess

Diese Richtlinie ermöglicht Vollzugriff auf alle Lambda-Aktionen. Es gewährt auch Berechtigungen für andere AWS Dienste, die zur Entwicklung und Wartung von Lambda-Ressourcen verwendet werden.

Die Richtlinie `AWSLambda_FullAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `lambda`: Ermöglicht Prinzipalen Vollzugriff auf Lambda.
- `cloudformation`— Ermöglicht Prinzipalen, AWS CloudFormation Stacks zu beschreiben und die Ressourcen in diesen Stacks aufzulisten.
- `cloudwatch`— Ermöglicht Prinzipalen, CloudWatch Amazon-Metriken aufzulisten und Metrikdaten abzurufen.
- `ec2`: Ermöglicht es Prinzipalen, Sicherheitsgruppen, Subnetze und VPCs zu beschreiben.
- `iam`: Ermöglicht es Prinzipalen, Richtlinien, Richtlinienversionen, Rollen, Rollenrichtlinien, angefügte Rollenrichtlinien und die Liste der Rollen abzurufen. Diese Richtlinie ermöglicht es Prinzipalen außerdem, Rollen an Lambda zu übergeben. Die Berechtigung `PassRole` wird verwendet, wenn Sie einer Funktion eine Ausführungsrolle zuweisen. Die `CreateServiceLinkedRole` Berechtigung wird beim Erstellen einer serviceverknüpften Rolle verwendet.
- `kms`— Ermöglicht Prinzipalen, Aliase aufzulisten und den Schlüssel für die Volumenverschlüsselung zu beschreiben.
- `logs`: Ermöglicht Prinzipalen das Beschreiben von Protokollstreams, das Abrufen und Filtern von Protokollereignissen sowie das Starten und Beenden von Live-Tail-Sitzungen.
- `states`— Ermöglicht es Prinzipalen, Zustandsmaschinen zu beschreiben und aufzulisten AWS Step Functions .
- `tag`: Ermöglicht es Prinzipalen, Ressourcen auf der Grundlage ihrer Tags abzurufen.
- `xray`— Ermöglicht Prinzipalen das Abrufen von AWS X-Ray Trace-Zusammenfassungen und das Abrufen einer nach ID spezifizierten Liste von Traces.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter AWS Lambda_FullAccess im AWS Managed Policy Reference Guide.

AWS verwaltete Richtlinie: AWS Lambda_ReadOnlyAccess

Diese Richtlinie gewährt schreibgeschützten Zugriff auf Lambda-Ressourcen und andere AWS Dienste, die zur Entwicklung und Wartung von Lambda-Ressourcen verwendet werden.

Die Richtlinie `AWSLambda_ReadOnlyAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `lambda`: Ermöglicht es Prinzipalen, alle Ressourcen abzurufen und aufzulisten.
- `cloudformation`— Ermöglicht Prinzipalen, Stacks zu beschreiben und aufzulisten und die Ressourcen in AWS CloudFormation diesen Stacks aufzulisten.
- `cloudwatch`— Ermöglicht Prinzipalen, CloudWatch Amazon-Metriken aufzulisten und Metrikdaten abzurufen.
- `ec2`: Ermöglicht es Prinzipalen, Sicherheitsgruppen, Subnetze und VPCs zu beschreiben.
- `iam`: Ermöglicht es Prinzipalen, Richtlinien, Richtlinienversionen, Rollen, Rollenrichtlinien, angefügte Rollenrichtlinien und die Liste der Rollen abzurufen.
- `kms`: Ermöglicht Prinzipalen das Auflisten von Aliasen.
- `logs`: Ermöglicht Prinzipalen das Beschreiben von Protokollstreams, das Abrufen und Filtern von Protokollereignissen sowie das Starten und Beenden von Live-Tail-Sitzungen.
- `states`— Ermöglicht es Prinzipalen, AWS Step Functions Zustandsmaschinen zu beschreiben und aufzulisten.
- `tag`: Ermöglicht es Prinzipalen, Ressourcen auf der Grundlage ihrer Tags abzurufen.
- `xray`— Ermöglicht Prinzipalen das Abrufen von AWS X-Ray Trace-Zusammenfassungen und das Abrufen einer nach ID spezifizierten Liste von Traces.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter AWS Lambda_ReadOnlyAccess im AWS Managed Policy Reference Guide.

AWS verwaltete Richtlinie: AWSLambdaBasicExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Hochladen von Protokollen in CloudWatch Logs.

Die Richtlinie `AWSLambdaBasicExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaBasicExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaBasicDurableExecutionRolePolicy

Diese Richtlinie gewährt Schreibberechtigungen für CloudWatch Protokolle und read/write Berechtigungen für langlebige Ausführungs-APIs, die von dauerhaften Lambda-Funktionen verwendet werden. Diese Richtlinie stellt die grundlegenden Berechtigungen bereit, die für dauerhafte Lambda-Funktionen erforderlich sind, die dauerhafte Ausführungs-APIs verwenden, um den Fortschritt aufrechtzuerhalten und den Status bei Funktionsaufrufen aufrechtzuerhalten.

Die Richtlinie `AWSLambdaBasicDurableExecutionRolePolicy` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `logs`— Ermöglicht es Prinzipalen, Protokollgruppen und Protokollstreams zu erstellen und Protokollereignisse in Logs zu schreiben. CloudWatch
- `lambda`— Ermöglicht es Prinzipalen, den dauerhaften Ausführungsstatus zu überprüfen und den dauerhaften Ausführungsstatus für langlebige Lambda-Funktionen abzurufen.

Weitere Einzelheiten zu dieser Richtlinie, einschließlich der neuesten Version des JSON-Richtliniendokuments, finden Sie unter AWSLambdaBasicDurableExecutionRolePolicy im AWS Referenzhandbuch für verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaDynamoDBExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Lesen von Datensätzen aus einem Amazon DynamoDB DynamoDB-Stream und zum Schreiben in CloudWatch Protokolle.

Die Richtlinie `AWSLambdaDynamoDBExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaDynamoDBExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaENIManagementAccess

Diese Richtlinie gewährt Berechtigungen zum Erstellen, Beschreiben und Löschen von Elastic Network-Schnittstellen, die von einer VPC-enabled Lambda-Funktion verwendet werden.

Die Richtlinie `AWSLambdaENIManagementAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaENIManagementAccessim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaInvocation-DynamoDB

Diese Richtlinie gewährt Lesezugriff auf Amazon DynamoDB Streams.

Die Richtlinie `AWSLambdaInvocation-DynamoDB` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaInvocation-DynamoDBim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaKinesisExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Lesen von Ereignissen aus einem Amazon Kinesis Kinesis-Datenstream und zum Schreiben in CloudWatch Protokolle.

Die Richtlinie `AWSLambdaKinesisExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaKinesisExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaMSKExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Lesen und Zugreifen auf Datensätze aus einem Amazon Managed Streaming for Apache Kafka Kafka-Cluster, zur Verwaltung elastischer Netzwerkschnittstellen und zum Schreiben in CloudWatch Protokolle.

Die Richtlinie `AWSLambdaMSKExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaMSKExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaRole

Diese Richtlinie erteilt Berechtigungen zum Aufrufen von Lambda-Funktionen.

Die Richtlinie `AWSLambdaRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaSQSQueueExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Lesen und Löschen von Nachrichten aus einer Amazon Simple Queue Service-Warteschlange sowie Schreibberechtigungen für CloudWatch Logs.

Die Richtlinie `AWSLambdaSQSQueueExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaSQSQueueExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaVPCAccessExecutionRole

Diese Richtlinie gewährt Berechtigungen zur Verwaltung elastischer Netzwerkschnittstellen innerhalb einer Amazon Virtual Private Cloud und zum Schreiben in CloudWatch Logs.

Die Richtlinie `AWSLambdaVPCAccessExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaVPCAccessExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaManagedEC2ResourceOperator

Diese Richtlinie ermöglicht automatisiertes Amazon Elastic Compute Cloud-Instanzmanagement und CloudWatch Amazon-Protokollierung für Lambda-Kapazitätsanbieter. Es gewährt Lambda die Erlaubnis, in Ihrem Namen Instance-Lebenszyklusvorgänge durchzuführen und Logs zu schreiben.

Die Richtlinie `AWSLambdaManagedEC2ResourceOperator` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `ec2:RunInstances`— Ermöglicht Lambda den Start neuer Amazon EC2 EC2-Instances unter der Bedingung, dass ec2: scaler.lambda.amazonaws.com ManagedResourceOperator entspricht und beschränkt die AMI-Nutzung nur auf Images. Amazon-owned
- `ec2:DescribeInstances`und `ec2:DescribeInstanceStatus` — Ermöglicht Lambda, den Instanzstatus zu überwachen und Instanzinformationen abzurufen.
- `ec2:CreateTags`— Ermöglicht Lambda, Amazon EC2 EC2-Ressourcen zu Verwaltungs- und Identifikationszwecken zu kennzeichnen.
- `ec2:DescribeAvailabilityZones`— Ermöglicht Lambda, verfügbare Zonen einzusehen, um Entscheidungen zur Instanzplatzierung zu treffen.
- `ec2:DescribeCapacityReservations`— Ermöglicht Lambda, Kapazitätsreservierungen für eine optimale Instance-Platzierung zu überprüfen.
- `ec2:DescribeInstanceTypes`und `ec2:DescribeInstanceTypeOfferings` — Ermöglicht Lambda, verfügbare Instance-Typen und deren Angebote zu überprüfen.
- `ec2:DescribeSubnets`— Ermöglicht Lambda, Subnetzkonfigurationen für die Netzwerkplanung zu untersuchen.
- `ec2:DescribeVpcEncryptionControls`— Ermöglicht Lambda, die Einstellungen der VPC-Verschlüsselungssteuerung für die Instanzkonfiguration zu untersuchen.
- `ec2:DescribeSecurityGroups`— Ermöglicht Lambda das Abrufen von Sicherheitsgruppeninformationen für die Konfiguration der Netzwerkschnittstelle.
- `ec2:CreateNetworkInterface`— Ermöglicht Lambda die Erstellung von Netzwerkschnittstellen und die Verwaltung von Subnetz- und Sicherheitsgruppenzuordnungen.
- `ec2:AttachNetworkInterface`— Ermöglicht Lambda das Anhängen von Netzwerkschnittstellen an Amazon EC2 EC2-Instances mit der Bedingung scaler.lambda.amazonaws.com`ec2:ManagedResourceOperator`.
- `logs:CreateLogGroup`— Ermöglicht Lambda die Erstellung von CloudWatch Logs-Protokollgruppen.
- `logs:CreateLogStream`und `logs:PutLogEvents` — Ermöglicht Lambda, Log-Streams zu erstellen und Log-Ereignisse in CloudWatch Logs zu schreiben.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaManagedEC2ResourceOperatorim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaServiceRolePolicy

Diese Richtlinie ist der serviceverknüpften Rolle zugeordnet, sodass Lambda 1) Instances, die als Teil von Lambda-Kapazitätsanbietern verwaltet werden, beenden und 2) Elastic Network Interfaces (ENIs) löschen kann, die als Teil von Lambda-Netzwerkkonnektoren verwaltet werden. AWSServiceRoleForLambda

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `ec2:TerminateInstances`— Ermöglicht Lambda, EC2-Instances unter der Bedingung zu beenden, dass ec2: ManagedResourceOperator gleich scaler.lambda.amazonaws.com ist.
- `ec2:DescribeInstanceStatus`und `ec2:DescribeInstances` — Ermöglicht Lambda die Beschreibung von EC2-Instances.
- `ec2:DescribeNetworkInterfacesec2:DescribeSubnets`, und `ec2:DescribeSecurityGroups` — Ermöglicht Lambda die Beschreibung von Netzwerkschnittstellen, Subnetzen und Sicherheitsgruppen.
- `ec2:DeleteNetworkInterface`— Ermöglicht Lambda, ENIs mit der Bedingung zu löschen, dass ec2: ManagedResourceOperator gleich network-connectors.lambda.amazonaws.com ist.

Weitere Informationen zu dieser Richtlinie finden Sie unter Verwenden von dienstverknüpften Rollen für Lambda.

AWS verwaltete Richtlinie: AWSLambdaNetworkConnectorOperatorPolicy

Diese Richtlinie gewährt Berechtigungen zur Erstellung und Verwaltung von elastic network interface (ENI) -Ressourcen, die vom Lambda Network Connector verwaltet werden.

Die Richtlinie `AWSLambdaNetworkConnectorOperatorPolicy` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `ec2:CreateNetworkInterface`— Erteilt die Berechtigung zum Erstellen von Netzwerkschnittstellen in einem beliebigen Subnetz mit zugehörigen Sicherheitsgruppen. Bei Netzwerkschnittstellenressourcen `aws:lambda:networkConnectorId` sind nur Tags mit den Schlüsseln `aws:lambda:networkConnectorName` und zulässig.
- `ec2:CreateTags`— Erteilt die Erlaubnis, Netzwerkschnittstellen bei der Erstellung zu kennzeichnen. Diese Berechtigung gilt nur, wenn die Aktion „Erstellen“ aktiviert ist `CreateNetworkInterface` und der Operator für verwaltete Ressourcen dies tut`network-connectors.lambda.amazonaws.com`.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaNetworkConnectorOperatorPolicyim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaInvokeWebFunctionEndpointAccess

Diese Richtlinie gewährt Berechtigungen zum Aufrufen von Lambda-Webfunktionsendpunkten.

Die Richtlinie `AWSLambdaInvokeWebFunctionEndpointAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaInvokeWebFunctionEndpointAccessim Referenzhandbuch für AWS verwaltete Richtlinien.

Lambda-Updates für AWS Verwaltete Richtlinien

Änderungen	Beschreibung	Date
AWSLambdaInvokeWebFunctionEndpointAccess – Neue Richtlinie	Lambda hat eine neue verwaltete Richtlinie hinzugefügt, um Berechtigungen zum Aufrufen von Lambda-Webfunktionsendpunkten zu gewähren.	1. Oktober 2026
AWSLambdaNetworkConnectorOperatorPolicy – Neue Richtlinie	Lambda hat eine neue verwaltete Richtlinie hinzugefügt, um Berechtigungen zur	19. Juni 2026

Änderungen	Beschreibung	Date
	Erstellung und Verwaltung von ENI-Ressourcen zu gewähren, die vom Lambda Network Connector verwaltet werden.	
AWSLambdaManagedEC2ResourceOperator— Veränderung	Lambda hat die `AWSLambdaManagedEC2ResourceOperator` Richtlinie aktualisiert, um die `logs:PutLogEvents` Aktionen `logs:CreateLogGroup` `logs:CreateLogStream` , und zuzulassen. Diese Berechtigungen ermöglichen es Lambda, Protokolle in Logs zu CloudWatch schreiben.	10. Juni 2026
AWSLambdaManagedEC2ResourceOperator— Veränderung	Lambda hat die `AWSLambdaManagedEC2ResourceOperator` Richtlinie aktualisiert, um die `ec2:DescribeVpcEncryptionControls` Aktion zuzulassen. Diese Berechtigung ermöglicht es Lambda, die Einstellungen der VPC-Verschlüsselungssteuerung für die Instanzkonfiguration zu überprüfen.	23. April 2026

Änderungen	Beschreibung	Date
AWSLambdaManagedEC2ResourceOperator – Neue Richtlinie	Lambda hat eine neue verwaltete Richtlinie hinzugefügt, um ein automatisiertes Amazon EC2 EC2-Instance-Management für Lambda-Kapazitätsanbieter zu ermöglichen, sodass der Scaler-Service Instance-Lebenszyklusoperationen durchführen kann.	30. November 2025
AWSLambdaServiceRolePolicy – Neue Richtlinie	Lambda hat eine neue verwaltete Richtlinie für die serviceverknüpfte Rolle hinzugefügt, damit Lambda Instanzen beenden kann, die als Teil von Lambda-Kapazitätsanbietern verwaltet werden.	30. November 2025
AWS Lambda_FullAccess — Veränderung	Lambda hat die `AWSLambda_FullAccess` Richtlinie aktualisiert, um die `iam:CreateServiceLinkedRole` Aktionen `kms:DescribeKey` und zuzulassen.	30. November 2025

Änderungen	Beschreibung	Date
AWSLambdaBasicDurableExecutionRolePolicy— Neue verwaltete Richtlinie	Lambda hat eine neue verwaltete Richtlinie veröffentlicht `AWSLambdaBasicDurableExecutionRolePolicy` , die Schreibberechtigungen für CloudWatch Protokolle und read/write Berechtigungen für langlebige Ausführungs-APIs bereitstellt, die von dauerhaften Lambda-Funktionen verwendet werden.	1. Dezember 2025
AWS Lambda_ReadOnlyAccess und AWS Lambda_FullAccess — Veränderung	Lambda hat die Richtlinien `AWSLambda_ReadOnlyAccess` und `AWSLambda_FullAccess` aktualisiert, um die Aktionen `logs:StartLiveTail` und `logs:StopLiveTail` zuzulassen.	17. März 2025
AWSLambdaVPCAccessExecutionRole— Veränderung	Lambda hat die `AWSLambdaVPCAccessExecutionRole` -Richtlinie aktualisiert, um die Aktion `ec2:DescribeSubnets` zuzulassen.	5. Januar 2024
AWS Lambda_ReadOnlyAccess — Veränderung	Lambda hat die `AWSLambda_ReadOnlyAccess` Richtlinie aktualisiert, sodass Prinzipale Stacks auflisten CloudFormation können.	27. Juli 2023

Änderungen	Beschreibung	Date
AWS Lambda hat begonnen, Änderungen zu verfolgen	AWS Lambda hat begonnen, Änderungen für die AWS verwalteten Richtlinien zu verfolgen.	27. Juli 2023

Fehlerbehebung AWS Lambda Identität und Zugriff

Diagnostizieren und beheben Sie mithilfe der folgenden Informationen gängige Probleme, die bei der Verwendung von Lambda und IAM auftreten können.

Themen

- Ich bin nicht autorisiert, eine Aktion in Lambda auszuführen.
- Ich bin nicht berechtigt, iam auszuführen: PassRole
- Ich möchte Personen außerhalb meiner Umgebung zulassen AWS-Konto um auf meine Lambda-Ressourcen zuzugreifen

Ich bin nicht autorisiert, eine Aktion in Lambda auszuführen.

Wenn Sie eine Fehlermeldung erhalten, dass Sie nicht zur Durchführung einer Aktion berechtigt sind, müssen Ihre Richtlinien aktualisiert werden, damit Sie die Aktion durchführen können.

Der folgende Beispielfehler tritt auf, wenn der IAM-Benutzer `mateojackson` versucht, über die Konsole Details zu einer fiktiven *`my-example-widget`*-Ressource anzuzeigen, jedoch nicht über `lambda:`*`GetWidget`*-Berechtigungen verfügt.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

In diesem Fall muss die Richtlinie für den Benutzer `mateojackson` aktualisiert werden, damit er mit der `lambda:`*`GetWidget`*-Aktion auf die *`my-example-widget`*-Ressource zugreifen kann.

Wenn Sie Hilfe benötigen, wenden Sie sich an Ihren AWS Administrator. Ihr Administrator hat Ihnen Ihre Anmeldeinformationen zur Verfügung gestellt.

Ich bin nicht berechtigt, iam auszuführen: PassRole

Wenn Sie die Fehlermeldung erhalten, dass Sie nicht zur Ausführung der Aktion „`iam:PassRole`“ autorisiert sind, müssen Ihre Richtlinien aktualisiert werden, um eine Rolle an Lambda übergeben zu können.

Einige AWS-Services ermöglichen es Ihnen, eine bestehende Rolle an diesen Dienst zu übergeben, anstatt eine neue Servicerolle oder eine dienstverknüpfte Rolle zu erstellen. Hierzu benötigen Sie Berechtigungen für die Übergabe der Rolle an den Dienst.

Der folgende Beispielfehler tritt auf, wenn ein IAM-Benutzer mit dem Namen `marymajor` versucht, die Konsole zu verwenden, um eine Aktion in Lambda auszuführen. Die Aktion erfordert jedoch, dass der Service über Berechtigungen verfügt, die durch eine Servicerolle gewährt werden. Mary besitzt keine Berechtigungen für die Übergabe der Rolle an den Dienst.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In diesem Fall müssen die Richtlinien von Mary aktualisiert werden, um die Aktion `iam:PassRole` ausführen zu können.

Wenn Sie Hilfe benötigen, wenden Sie sich an Ihren AWS Administrator. Ihr Administrator hat Ihnen Ihre Anmeldeinformationen zur Verfügung gestellt.

Ich möchte Personen außerhalb meiner Umgebung zulassen AWS-Konto um auf meine Lambda-Ressourcen zuzugreifen

Sie können eine Rolle erstellen, mit der Benutzer in anderen Konten oder Personen außerhalb Ihrer Organisation auf Ihre Ressourcen zugreifen können. Sie können festlegen, wem die Übernahme der Rolle anvertraut wird. Im Fall von Diensten, die ressourcenbasierte Richtlinien oder Zugriffskontrolllisten (Access Control Lists, ACLs) verwenden, können Sie diese Richtlinien verwenden, um Personen Zugriff auf Ihre Ressourcen zu gewähren.

Weitere Informationen finden Sie hier:

- Informationen dazu, ob Lambda diese Funktionen unterstützt, finden Sie unter Wie AWS Lambda funktioniert mit IAM.
- Informationen dazu, wie Sie Zugriff auf Ihre Ressourcen gewähren können, AWS-Konten die Ihnen gehören, finden Sie im IAM-Benutzerhandbuch unter Gewähren des Zugriffs auf einen IAM-Benutzer in einem anderen AWS-Konto , dem Sie gehören.

- Informationen dazu, wie Sie Dritten Zugriff auf Ihre Ressourcen gewähren können AWS-Konten, finden Sie AWS-Konten im IAM-Benutzerhandbuch unter Gewähren des Zugriffs für Dritte.
- Informationen dazu, wie Sie über einen Identitätsverbund Zugriff gewähren, finden Sie unter Gewähren von Zugriff für extern authentifizierte Benutzer (Identitätsverbund) im IAM-Benutzerhandbuch.
- Informationen zum Unterschied zwischen der Verwendung von Rollen und ressourcenbasierten Richtlinien für den kontoübergreifenden Zugriff finden Sie unter Kontoübergreifender Ressourcenzugriff in IAM im IAM-Benutzerhandbuch.

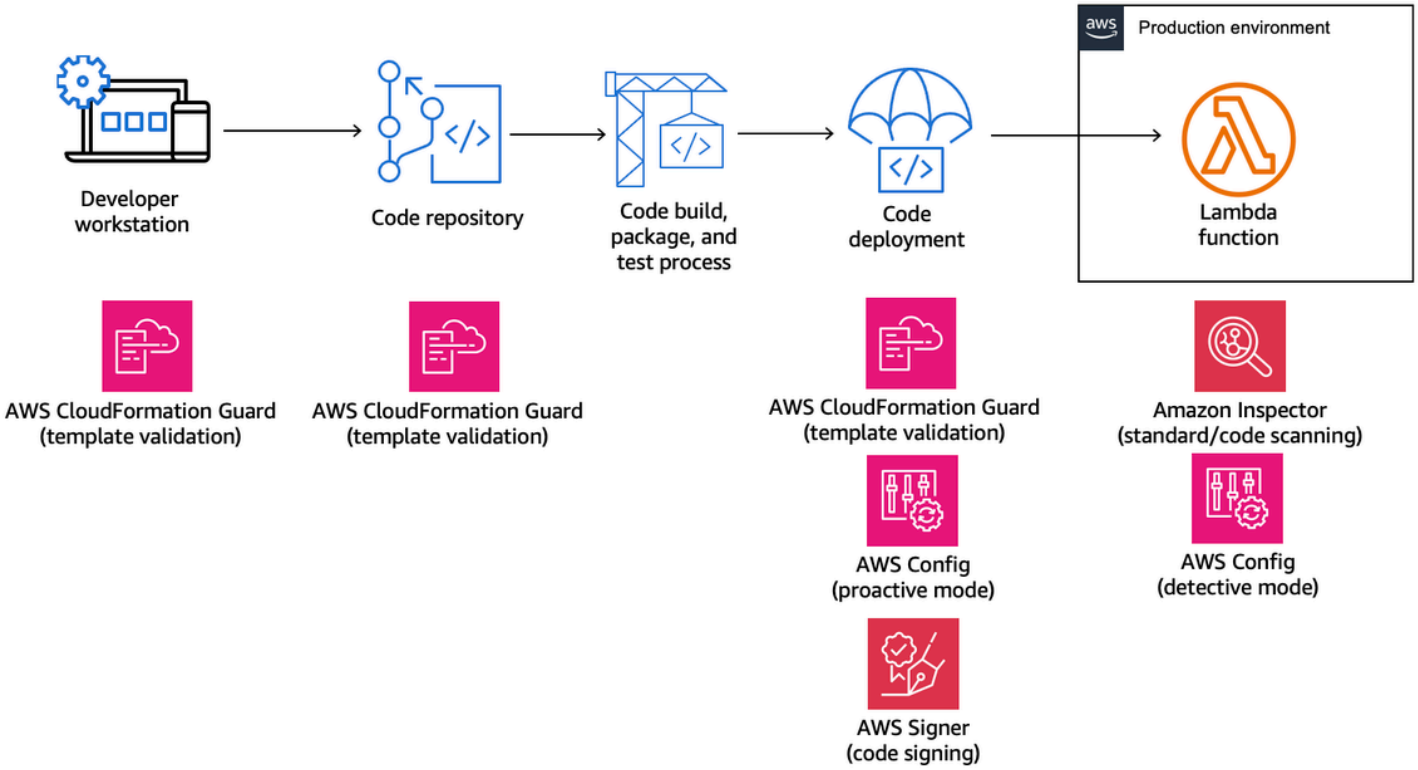
Erstellen Sie eine Governance-Strategie für Lambda-Funktionen und -Layer

Bei der Entwicklung und Bereitstellung cloudnativer Serverless-Anwendungen müssen Sie durch angemessene Governance- und Schutzmaßnahmen für Agilität und eine schnelle Markteinführung sorgen. Sie setzen Prioritäten auf Unternehmensebene und betonen vielleicht Agilität als oberste Priorität oder alternativ die Risikoaversion durch Governance, Leitplanken und Kontrollen. Realistisch betrachtet werden Sie keine „“ -Strategie haben, sondern eine „Undeither/or“ -Strategie, die sowohl Agilität als auch Leitplanken in Ihrem Softwareentwicklungszyklus in Einklang bringt. Ganz gleich, wo im Lebenszyklus Ihres Unternehmens diese Anforderungen stehen, Sie werden mit hoher Wahrscheinlichkeit Governance-Funktionen in Ihre Prozesse und Toolchains implementieren müssen.

Hier sind einige Beispiele für Governance-Kontrollen, die ein Unternehmen für Lambda implementieren könnte:

- Lambda-Funktionen dürfen nicht öffentlich zugänglich sein.
- Lambda-Funktionen müssen mit einer VPC verbunden sein.
- Lambda-Funktionen sollten keine veralteten Laufzeiten verwenden.
- Lambda-Funktionen müssen mit einer Reihe erforderlicher Tags markiert werden.
- Lambda-Layer dürfen außerhalb der Organisation nicht zugänglich sein.
- Wenn Lambda-Funktionen mit einer Sicherheitsgruppe verbunden sind, müssen beide übereinstimmende Tags haben.
- Lambda-Funktionen mit einem verbundenen Layer müssen eine genehmigte Version verwenden
- Lambda-Umgebungsvariablen müssen im Ruhezustand mit einem vom Kunden verwalteten Schlüssel verschlüsselt sein.

Das folgende Diagramm ist ein Beispiel für eine umfassende Governance-Strategie, bei der Kontrollen und Richtlinien während des gesamten Softwareentwicklungs- und Bereitstellungsprozesses implementiert werden:



In den folgenden Themen wird erklärt, wie Sie Kontrollen für die Entwicklung und Bereitstellung von Lambda-Funktionen in Ihrem Unternehmen implementieren. Die Ausführungen gelten für Startups und große Unternehmen gleichermaßen. Möglicherweise verfügt Ihr Unternehmen bereits über geeignete Tools. In den folgenden Themen wird ein modularer Ansatz für diese Kontrollen verfolgt, damit Sie die Komponenten auswählen können, die Sie tatsächlich benötigen. Zu den Themen gehörenthe section called “Proaktive Kontrollen mit Guard”,the section called “Scannen von Code”, the section called “Codesignatur” und. the section called “Beobachtbarkeit”

Themen

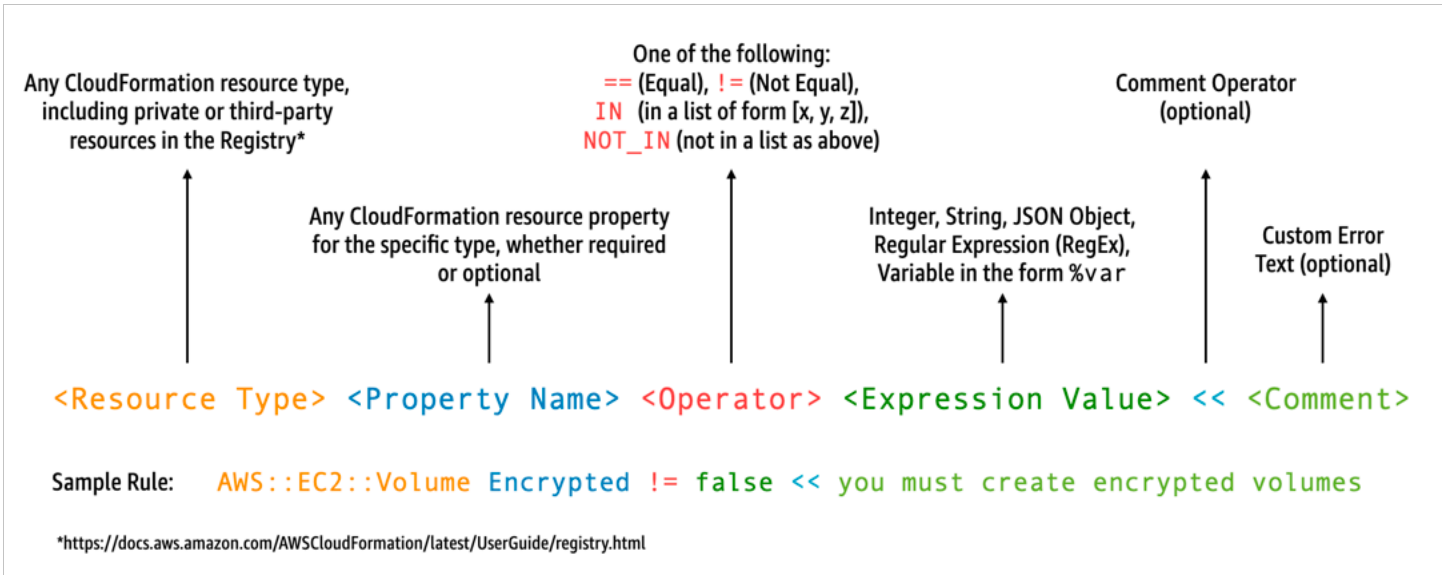
- Proaktive Kontrollen für Lambda mit AWS CloudFormation Guard
- Implementieren Sie präventive Kontrollen für Lambda mit AWS Config
- Erkennen Sie nicht konforme Lambda-Bereitstellungen und -Konfigurationen mit AWS Config
- Lambda-Codesignatur mit AWS Signer
- Automatisieren Sie Sicherheitsbewertungen für Lambda mit Amazon Inspector
- Implementieren von Beobachtbarkeit für Lambda-Sicherheit und -Compliance

Proaktive Kontrollen für Lambda mit AWS CloudFormation Guard

AWS CloudFormation Guard ist ein allgemein verwendbares Open-Source-Tool zur Policy-as-Code-Evaluierung. Es kann für präventive Governance und Compliance verwendet werden, indem Infrastructure as Code (IaC)-Vorlagen und Servicezusammenstellungen anhand von Richtlinienregeln validiert werden. Diese Regeln können an die Anforderungen Ihres Teams oder Ihrer Organisation angepasst werden. Für Lambda-Funktionen können die Guard-Regeln verwendet werden, um die Ressourcenerstellung und Konfigurationsupdates zu steuern. Dazu werden die erforderlichen Eigenschaftseinstellungen definiert, die beim Erstellen oder Aktualisieren einer Lambda-Funktion erforderlich sind.

Compliance-Administratoren definieren die Liste der Kontrollen und Governance-Richtlinien, die für die Bereitstellung und Aktualisierung von Lambda-Funktionen erforderlich sind. Plattformadministratoren implementieren die Kontrollen in CI/CD Pipelines, als Webhooks zur Pre-Commit-Validierung mit Code-Repositorys, und stellen Entwicklern Befehlszeilentools zur Validierung von Vorlagen und Code auf lokalen Arbeitsstationen zur Verfügung. Entwickler verfassen Code, validieren Vorlagen mit Befehlszeilentools und übertragen den Code dann in Repositorys, die dann vor der Bereitstellung in einer Umgebung automatisch über die CI/CD Pipelines validiert werden. AWS

Mit Guard können Sie Ihre Regeln schreiben und Ihre Steuerelemente wie folgt in einer domänenspezifischen Sprache implementieren.



Angenommen, Sie möchten, dass Entwickler nur die neuesten Laufzeiten verwenden. Sie könnten zwei verschiedene Richtlinien erstellen. Eine, um Laufzeiten zu bestimmen, die bereits veraltet sind,

und eine andere, um Laufzeiten zu bestimmen, die demnächst veraltet sein werden. Dazu könnten Sie die folgende `etc/rules.guard`-Datei schreiben:

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

Nehmen wir nun an, Sie schreiben die folgende `iac/lambda.yaml` CloudFormation Vorlage, die eine Lambda-Funktion definiert:

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
```

```
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

Nach der Installation des Guard-Dienstprogramms überprüfen Sie die Vorlage:

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

Das Ergebnis sieht folgendermaßen aus:

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
       ComparisonError {
         Message          = Lambda function is using a runtime that is targeted for
 deprecation.
         Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
       }
         PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
         Operator        = NOT IN
         Value           = "python3.7"
         ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
         Code:
              86.  Fn:
              87.    Type: AWS::Lambda::Function
              88.    Properties:
              89.      Runtime: python3.7
              90.      CodeUri: src
              91.      Handler: fn.handler

    }
   }
```

```
  }
}
```

Mit Guard können Ihre Entwickler von ihren lokalen Entwickler-Workstations aus sehen, dass sie die Vorlage aktualisieren müssen, um eine von der Organisation zugelassene Laufzeit zu verwenden. Dies geschieht, bevor ein Commit in ein Code-Repository übertragen wird und anschließend Prüfungen innerhalb einer CI/CD Pipeline fehlschlagen. Ihre Entwickler erhalten direktes Feedback, wie sie konforme Vorlagen entwickeln und können ihre Zeit darauf verwenden, Code zu schreiben, der geschäftlichen Nutzen bietet. Dieses Steuerelement kann auf der lokalen Entwickler-Workstation in einem Webhook zur Validierung vor dem Commit and/or in der CI/CD Pipeline vor der Bereitstellung angewendet werden.

Einschränkungen

Wenn Sie AWS Serverless Application Model (AWS SAM) -Vorlagen verwenden, um Lambda-Funktionen zu definieren, beachten Sie, dass Sie die Guard-Regel wie folgt aktualisieren müssen, um nach dem `AWS::Serverless::Function` Ressourcentyp zu suchen.

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

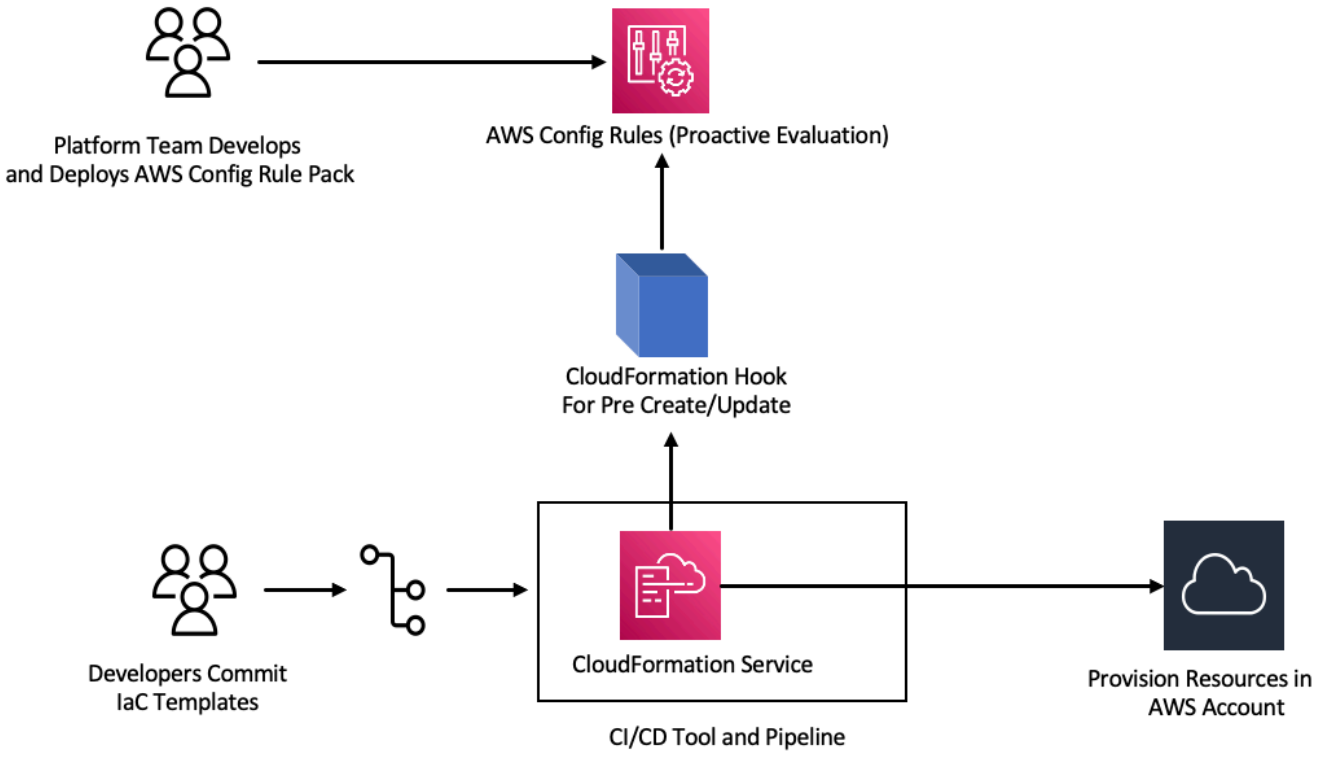
Guard erwartet außerdem, dass die Eigenschaften in der Ressourcendefinition enthalten sind. In der Zwischenzeit ermöglichen AWS SAM Vorlagen die Angabe von Eigenschaften in einem separaten Globals-Abschnitt. Eigenschaften, die im Globals-Abschnitt definiert sind, werden nicht mit Ihren Guard-Regeln validiert.

Beachten Sie, dass Guard, wie in der Dokumentation zur Fehlerbehebung bei Guard beschrieben, keine intrinsischen Kurzformen wie `!GetAtt` oder `!Sub` unterstützt, sondern stattdessen die Verwendung der erweiterten Formen erfordert: `Fn::GetAtt` und `Fn::Sub`. (Im vorherigen Beispiel wird die Role-Eigenschaft nicht ausgewertet, daher wurde der Einfachheit halber die intrinsische Kurzform verwendet.)

Implementieren Sie präventive Kontrollen für Lambda mit AWS Config

Es ist wichtig, so früh wie möglich im Entwicklungsprozess die Konformität Ihrer Serverless-Anwendungen sicherzustellen. In diesem Thema behandeln wir die Implementierung präventiver Kontrollen mithilfe von AWS Config. Mit AWS Config können Sie Konformitätsprüfungen zu einem früheren Zeitpunkt im Entwicklungsprozess implementieren und dieselben Kontrollen in Ihren Pipelines implementieren. CI/CD Dadurch werden auch Ihre Kontrollen in einem zentral verwalteten Regelarchiv standardisiert, sodass Sie Ihre Kontrollen konsistent auf alle Ihre Konten anwenden können. AWS

Nehmen wir zum Beispiel an, Ihre Compliance-Administratoren haben eine Anforderung definiert, um sicherzustellen, dass alle Lambda-Funktionen Tracing beinhalten AWS X-Ray . Im proaktiven Modus können Sie vor der Bereitstellung Konformitätsprüfungen für Ihre Lambda-Funktionsressourcen durchführen. So reduzieren Sie das Risiko, falsch konfigurierte Lambda-Funktionen bereitzustellen, und Entwickler sparen Zeit, indem Sie ihnen schnelleres Feedback zur Infrastruktur in Form von Codevorlagen geben. AWS Config Im Folgenden finden Sie eine Visualisierung des Ablaufs präventiver Kontrollen mit: AWS Config



Nehmen wir an, es wird festgelegt, dass für alle Lambda-Funktionen die Ablaufverfolgung aktiviert sein muss. Als Reaktion darauf stellt das Plattformteam fest, dass eine bestimmte AWS Config Regel proaktiv für alle Konten gelten muss. Diese Regel kennzeichnet jede Lambda-Funktion, für die keine konfigurierte X-Ray Ablaufverfolgungskonfiguration vorhanden ist, als nicht konforme Ressource. Das Team entwickelt eine Regel, verpackt sie in einem Konformitätspaket und verteilt das Konformitätspaket für alle Konten, um sicherzustellen, dass alle AWS Konten im Unternehmen diese Kontrollen einheitlich anwenden. Sie können die Regel in der AWS CloudFormation Guard 2.x.x-Syntax schreiben, die folgende Form hat:

```
rule name when condition { assertion }
```

Im Folgenden finden Sie ein Beispiel für eine Guard-Regel, die überprüft, ob für Lambda-Funktionen die Ablaufverfolgung aktiviert ist:

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

Das Plattformteam ergreift weitere Maßnahmen, indem es vorschreibt, dass bei jeder AWS CloudFormation Bereitstellung ein Pre-Hook aufgerufen wird. create/update https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/hooks-structure.html Das Team übernimmt die volle Verantwortung für die Entwicklung dieses Hooks und die Konfiguration der Pipeline, mit dem Ziel, die zentrale Kontrolle der Compliance-Regeln zu stärken und ihre konsistente Anwendung in allen Implementierungen sicherzustellen. Informationen zum Entwickeln, Verpacken und Registrieren eines Hooks finden Sie unter Entwickeln von AWS CloudFormation Hooks in der CloudFormation Dokumentation zur Befehlszeilenschnittstelle (CFN-CLI). Sie können die CloudFormation CLI verwenden, um das Hook-Projekt zu erstellen:

```
cfn init
```

Der Befehl fragt Sie nach einigen grundlegenden Informationen zu Ihrem Hook-Projekt und erstellt ein Projekt mit den folgenden Dateien:

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
```

```
template.yml
hook-role.yaml
```

Der Hook-Entwickler fügt der Konfigurationsdatei `<hook-name>.json` den gewünschten Zielressourcentyp hinzu. In der folgenden Konfiguration wird ein Hook so konfiguriert, dass er ausgeführt wird, bevor eine Lambda-Funktion mit CloudFormation erstellt wird. Sie können ähnliche Handler auch für die Aktionen `preUpdate` und `preDelete` hinzufügen.

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

Sie müssen außerdem sicherstellen, dass der CloudFormation Hook über die entsprechenden Berechtigungen zum Aufrufen der AWS Config APIs verfügt. Aktualisieren Sie dazu die Rollendefinitionsdatei `hook-role.yaml`. Die Rollendefinitionsdatei hat standardmäßig die folgende Vertrauensrichtlinie, die es ermöglicht, die Rolle CloudFormation zu übernehmen.

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

Damit der Hook Konfigurations-APIs aufrufen kann, müssen Sie der Policy-Anweisung die folgenden Berechtigungen hinzufügen. Dann reichen Sie das Hook-Projekt mit dem `cfn submit` Befehl ein, mit dem eine Rolle mit den erforderlichen Berechtigungen für Sie CloudFormation erstellt wird.

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
```

```
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

Als Nächstes müssen Sie eine Lambda-Funktion in eine `src/handler.py`-Datei schreiben. In dieser Datei finden Sie Methoden mit dem Namen und `preCreate`, `preUpdate` und `preDelete`, die bereits bei der Initiierung des Projekts erstellt wurden. Ihr Ziel ist es, eine gemeinsame, wiederverwendbare Funktion zu schreiben, die die AWS Config `StartResourceEvaluation` API im proaktiven Modus aufruft, indem Sie die verwenden AWS SDK für Python (Boto3). Dieser API-Aufruf verwendet Ressourceneigenschaften als Eingabe und vergleicht die Ressource mit der Regeldefinition.

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

Jetzt können Sie die allgemeine Funktion vom Handler für den Pre-Create-Hook aus aufrufen. Ein Beispiel für den Handler:

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

Nach diesem Schritt können Sie den Hook registrieren und ihn so konfigurieren, dass er alle Ereignisse zur AWS Lambda Funktionserstellung abhört.

Ein Entwickler bereitet die IaC-Vorlage (Infrastructure as Code) für einen Serverless-Microservice mithilfe von Lambda vor. Diese Vorbereitung umfasst die Einhaltung interner Standards, gefolgt von lokalen Tests und dem Commit der Vorlage in das Repository. Hier ist ein Beispiel für eine IaC-Vorlage:

```
  MyLambdaFunction:
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json

        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.14
    TracingConfig:
```

```
        Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

Als Teil des CI/CD Prozesses ruft der CloudFormation Dienst bei der Bereitstellung der CloudFormation Vorlage den create/update Pre-Hook direkt vor der Bereitstellung des `AWS::Lambda::Function` Ressourcentyps auf. Der Hook verwendet AWS Config Regeln, die im proaktiven Modus ausgeführt werden, um zu überprüfen, ob die Lambda-Funktionskonfiguration die vorgeschriebene Tracing-Konfiguration enthält. Die Antwort des Hooks bestimmt den nächsten Schritt.

Wenn die Anforderungen erfüllt sind, signalisiert der Hook Erfolg und CloudFormation fährt mit der Bereitstellung der Ressourcen fort. Andernfalls schlägt die CloudFormation Stack-Bereitstellung fehl, die Pipeline kommt sofort zum Stillstand und das System zeichnet die Details zur späteren Überprüfung auf. An die relevanten Stakeholder werden Compliance-Benachrichtigungen gesendet.

Sie finden die success/fail Hook-Informationen in der CloudFormation Konsole:

Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

Wenn Sie Logs für Ihren CloudFormation Hook aktiviert haben, können Sie das Ergebnis der Hook-Bewertung erfassen. Hier ist ein Beispielprotokoll für einen Hook mit dem Status „Fehlgeschlagen“, was darauf hinweist, dass die Lambda-Funktion nicht X-Ray aktiviert ist:

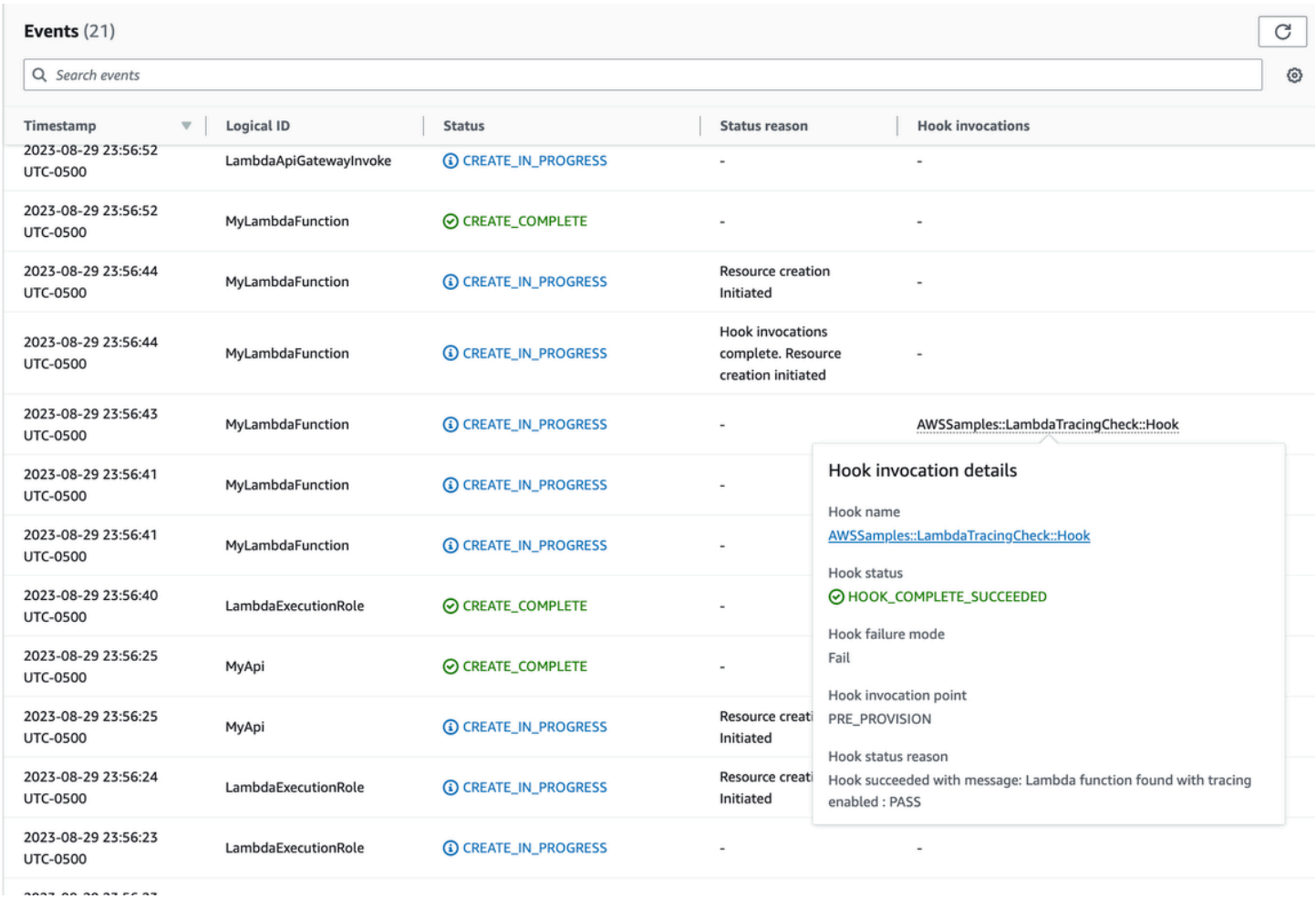
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

Hat der Entwickler entschieden, die IaC so zu ändern, dass der Wert `TracingConfig Mode` in `Active` aktualisiert wird und die Bereitstellung erneut erfolgt, wird der Hook erfolgreich ausgeführt und der Stack fährt mit der Erstellung der Lambda-Ressource fort.



Auf diese Weise können Sie präventive Kontrollen AWS Config im proaktiven Modus implementieren, wenn Sie serverlose Ressourcen in Ihren Konten entwickeln und bereitstellen. AWS Durch die Integration von AWS Config Regeln in die CI/CD Pipeline können Sie nicht konforme Ressourcenbereitstellungen identifizieren und optional blockieren, z. B. Lambda-Funktionen, für die keine aktive Ablaufverfolgungskonfiguration vorhanden ist. Dadurch wird sichergestellt, dass nur Ressourcen in Ihren Umgebungen bereitgestellt werden, die den neuesten Governance-Richtlinien entsprechen. AWS

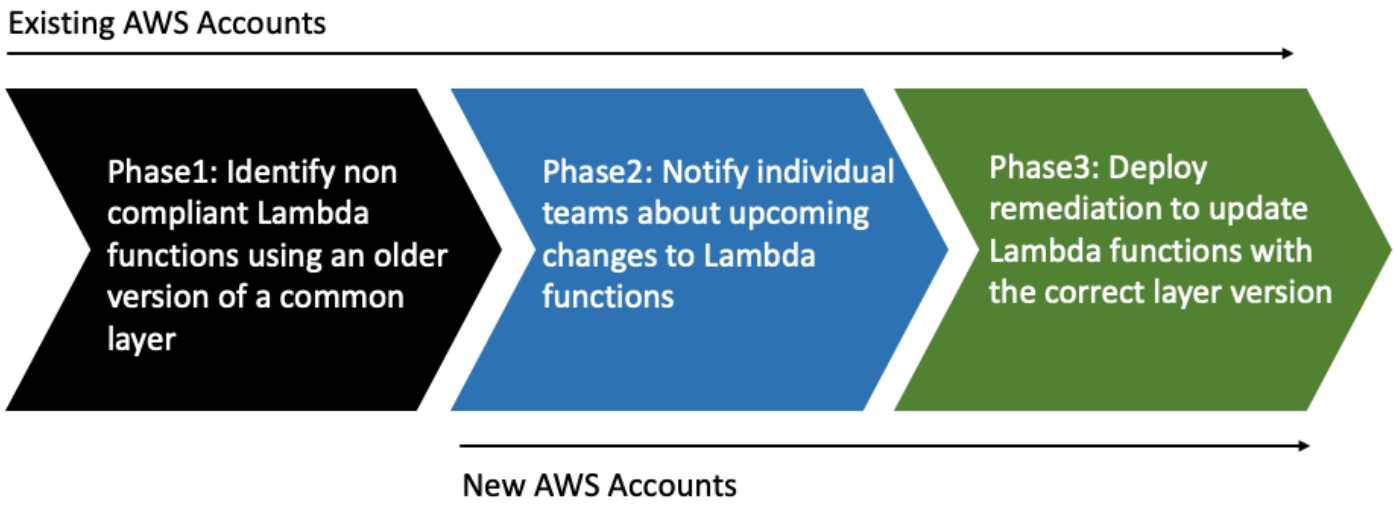
Erkennen Sie nicht konforme Lambda-Bereitstellungen und -Konfigurationen mit AWS Config

Zusätzlich zur proaktiven Bewertung AWS Config können Sie auch reaktiv Ressourceneinsätze und Konfigurationen erkennen, die nicht Ihren Unternehmensrichtlinien entsprechen. Das ist wichtig, weil sich Governance-Richtlinien ändern können, wenn Ihr Unternehmen sich weiterentwickelt und neue Best-Practices eingeführt werden.

Angenommen, Sie legen bei der Bereitstellung oder Aktualisierung von Lambda-Funktionen eine brandneue Richtlinie fest: Alle Lambda-Funktionen müssen immer eine bestimmte, genehmigte Lambda-Layer-Version verwenden. Sie können AWS Config konfigurieren, um neue oder aktualisierte Funktionen für Layer-Konfigurationen zu überwachen. Wenn eine Funktion AWS Config erkannt wird, die keine genehmigte Layer-Version verwendet, kennzeichnet sie die Funktion als nicht konforme Ressource.

Sie können optional so konfigurieren AWS Config , dass die Ressource automatisch repariert wird, indem Sie mithilfe eines Automatisierungsdokuments eine Behebungsaktion angeben. AWS Systems Manager Sie könnten beispielsweise ein Automatisierungsdokument in Python mit dem schreiben, das die nicht konforme Funktion so aktualisiert AWS SDK für Python (Boto3), dass sie auf die genehmigte Layer-Version verweist. Es AWS Config dient somit sowohl als Detektiv- als auch als Korrekturkontrolle und automatisiert das Compliance-Management.

Wir brechen diesen Prozess in drei zentrale Implementierungsphasen auf:



Phase 1: Identifizieren der Zugriffsressourcen

Aktivieren Sie zunächst AWS Config alle Ihre Konten und konfigurieren Sie es so, dass es AWS Lambda-Funktionen aufzeichnet. Auf diese Weise können AWS Config Sie beobachten, wann Lambda-Funktionen erstellt oder aktualisiert werden. Anschließend können Sie benutzerdefinierte Richtlinienregeln konfigurieren, um nach bestimmten Richtlinienverstößen zu suchen, die AWS CloudFormation Guard -Syntax verwenden. Guard-Regeln haben die folgende allgemeine Form:

```
rule name when condition { assertion }
```

Im Folgenden finden Sie eine Beispielregel, mit der überprüft wird, ob ein Layer auf eine alte Layer-Version eingestellt ist:

```
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

Sehen wir uns die Syntax und den Regelaufbau etwas genauer an:

- Regelname: Der Name der Regel im Beispiel lautet `desiredlayer`.
- Bedingung: Diese Klausel gibt die Bedingung an, unter der die Regel überprüft werden soll. Im angegebenen Beispiel lautet die Bedingung `configuration.layers !empty`. Das bedeutet, dass die Ressource nur ausgewertet werden sollte, wenn die `layers`-Eigenschaft in der Konfiguration nicht leer ist.
- Assertion: Nach der `when`-Klausel bestimmt die Assertion, was die Regel überprüft. Die Assertion (Behauptung) `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` prüft, ob einer der ARNs des Lambda-Layers nicht mit dem Wert `OldLayerArn` übereinstimmt. Wenn es keine Übereinstimmung gibt, ist die Assertion wahr und die Regel gilt als bestanden. Andernfalls schlägt sie fehl.

`CONFIG_RULE_PARAMETERS`ist ein spezieller Satz von Parametern, der mit der AWS Config Regel konfiguriert wird. In diesem Fall ist `OldLayerArn` ein Parameter in `CONFIG_RULE_PARAMETERS`. Benutzer können so einen bestimmten ARN-Wert als veraltet oder abgelaufen festlegen. Anschließend überprüft die Regel, ob Lambda-Funktionen diesen alten ARN verwenden.

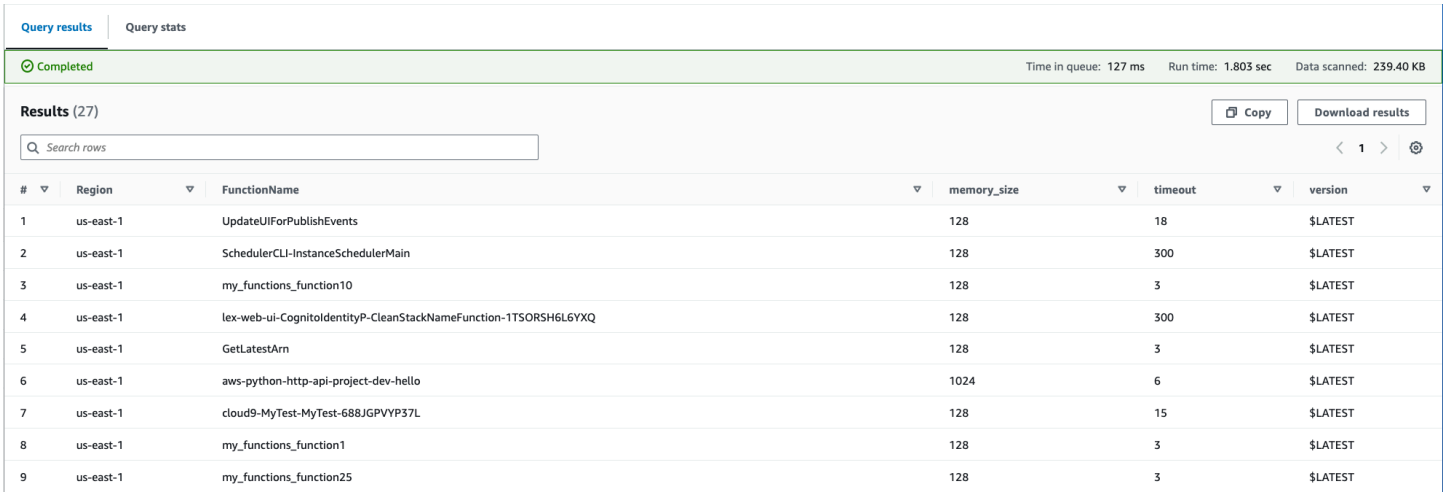
Phase 2: Visualisierung und Entwicklung

AWS Config sammelt Konfigurationsdaten und speichert diese Daten in Amazon Simple Storage Service (Amazon S3) -Buckets. Sie können Amazon Athena verwenden, um diese Daten direkt aus Ihren S3-Buckets abzufragen. Mit Athena können Sie diese Daten auf Organisationsebene aggregieren und sich einen ganzheitlichen Überblick über die Ressourcenkonfigurationen in all Ihren Konten verschaffen. Informationen zum Einrichten der Aggregation von Ressourcenkonfigurationsdaten finden Sie unter Visualisieren von AWS Config Daten mit Athena und Amazon Quick im Blog AWS Cloud Operations and Management.

Im Folgenden finden Sie ein Beispiel für eine Athena-Abfrage zur Identifizierung aller Lambda-Funktionen, die einen bestimmten Layer-ARN verwenden:

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

Hier sind die Ergebnisse der Abfrage:



Query results | Query stats

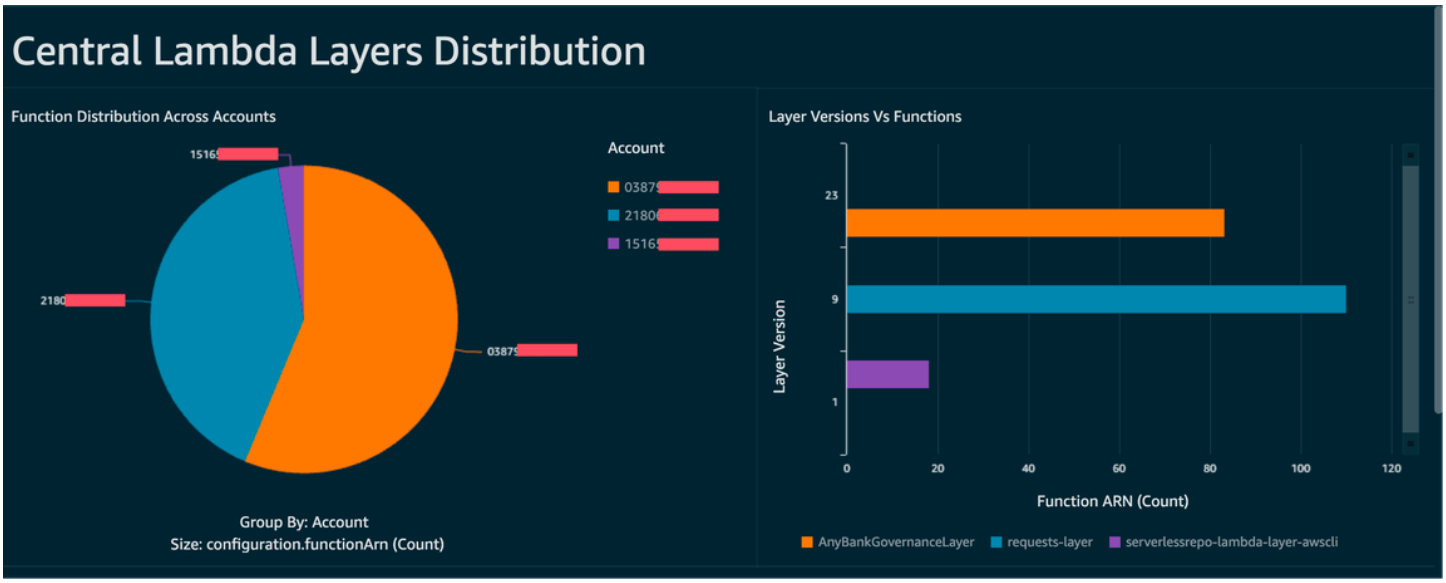
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

Results (27) — Copy — Download results

Search rows — 1

#	Region	FunctionName	memory_size	timeout	version
1	us-east-1	UpdateUIForPublishEvents	128	18	$LATEST
2	us-east-1	SchedulerCLI-InstanceSchedulerMain	128	300	$LATEST
3	us-east-1	my_functions_function10	128	3	$LATEST
4	us-east-1	lex-web-ui-CognitoIdentityP-CleanStackNameFunction-1TSORSH6L6YXQ	128	300	$LATEST
5	us-east-1	GetLatestArn	128	3	$LATEST
6	us-east-1	aws-python-http-api-project-dev-hello	1024	6	$LATEST
7	us-east-1	cloud9-MyTest-MyTest-688JGPVYP37L	128	15	$LATEST
8	us-east-1	my_functions_function1	128	3	$LATEST
9	us-east-1	my_functions_function25	128	3	$LATEST

Nachdem die AWS Config Daten unternehmensweit aggregiert sind, können Sie dann mithilfe von Amazon Quick ein Dashboard erstellen. https://aws.amazon.com/quicksight/ Indem Sie Ihre Athena-Ergebnisse in Quick importieren, können Sie sich vorstellen, wie gut Ihre Lambda-Funktionen die Layer-Versionsregel einhalten. Im Dashboard können Sie konforme und nicht konforme Ressourcen hervorheben und dementsprechend Durchsetzungsrichtlinien festlegen, wie im nächsten Abschnitt beschrieben. Die folgende Abbildung zeigt ein Beispiel-Dashboard, in dem die Verteilung der Layer-Versionen auf Funktionen innerhalb der Organisation aufgeschlüsselt ist.



Phase 3: Implementierung und Durchsetzung

Sie können jetzt optional Ihre Layer-Versionsregel, die Sie in Phase 1 erstellt haben, mithilfe eines Systems Manager-Automatisierungsdokuments, das Sie als Python-Skript erstellen,

mit einer Korrekturmaßnahme kombinieren. AWS SDK für Python (Boto3) Das Skript ruft die UpdateFunctionConfiguration API-Aktion für jede Lambda-Funktion auf und aktualisiert die Funktionskonfiguration mit dem neuen Layer-ARN. Alternativ können Sie das Skript so schreiben, dass eine Pull-Anfrage an das Code-Repository gesendet wird, um den Layer-ARN zu aktualisieren. Auf diese Weise werden künftige Codebereitstellungen ebenfalls mit dem richtigen Layer-ARN aktualisiert.

Lambda-Codesignatur mit AWS Signer

AWS Signerist ein vollständig verwalteter Codesignaturdienst, mit dem Sie Ihren Code anhand einer digitalen Signatur validieren können, um zu bestätigen, dass der Code unverändert ist und von einem vertrauenswürdigen Herausgeber stammt. AWS Signer kann in Verbindung mit verwendet werden, AWS Lambda um vor der Bereitstellung in Ihren Umgebungen zu überprüfen, ob Funktionen und Ebenen unverändert sind. AWS Ihr Unternehmen bleibt so vor böswilligen Akteuren geschützt, die sich möglicherweise Zugangsdaten verschafft haben, um neue Funktionen zu entwickeln oder bestehende Funktionen zu verändern.

Um die Codesignatur für Ihre Lambda-Funktionen einzurichten, erstellen Sie zunächst einen S3-Bucket mit aktivierter Versionierung. Erstellen Sie anschließend ein Signaturprofil mit AWS Signer, geben Sie Lambda als Plattform an und geben Sie dann einen Zeitraum von Tagen an, in dem das Signaturprofil gültig ist. Beispiel:

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

Verwenden Sie dann das Signaturprofil und erstellen Sie eine Signaturkonfiguration mit Lambda. Legen Sie fest, was geschehen soll, wenn in der Signaturkonfiguration ein Artefakt erkannt wird, das nicht mit der erwarteten digitalen Signatur übereinstimmt: warnen (aber die Bereitstellung zulassen) oder erzwingen (und die Bereitstellung blockieren). Das folgende Beispiel ist für Erzwingen und Bereitstellungen blockieren konfiguriert.

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

Sie haben jetzt Lambda AWS Signer so konfiguriert, dass nicht vertrauenswürdige Bereitstellungen blockiert werden. Nehmen wir an, Sie haben eine Feature-Anfrage fertig programmiert und sind nun

bereit, die Funktion bereitzustellen. Der erste Schritt besteht darin, den Code mit den entsprechenden Abhängigkeiten zu komprimieren und dann das Artefakt mit dem von Ihnen erstellten Signaturprofil zu signieren. Sie können dies tun, indem Sie das ZIP-Artefakt in den S3-Bucket hochladen und dann einen Signaturauftrag starten.

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

Sie erhalten eine Ausgabe wie folgt, wobei das das Objekt `jobId` ist, das im Ziel-Bucket und Präfix erstellt wurde, und die 12-stellige AWS-Konto ID `jobOwner` ist, unter der der Job ausgeführt wurde.

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

Jetzt können Sie Ihre Funktion mithilfe des signierten S3-Objekts und der von Ihnen erstellten Codesignatur-Konfiguration bereitstellen.

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

Sie können eine Funktionsbereitstellung alternativ mit dem ursprünglichen unsignierten Quell-ZIP-Artefakt testen. Die Bereitstellung sollte mit der folgenden Fehlermeldung fehlschlagen:

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

Wenn Sie Ihre Funktionen mithilfe von AWS Serverless Application Model (AWS SAM) erstellen und bereitstellen, übernimmt der Package-Befehl das Hochladen des ZIP-Artefakts auf S3. Außerdem

wird der Signierauftrag gestartet und das signierte Artefakt abgerufen. Verwenden Sie dazu den folgenden Befehl mit diesen Parametern:

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer hilft Ihnen dabei, zu überprüfen, ob Zip-Artefakte, die in Ihren Konten bereitgestellt werden, für die Bereitstellung vertrauenswürdig sind. Sie können den oben genannten Prozess in Ihre CI/CD Pipelines aufnehmen und verlangen, dass allen Funktionen eine Codesignaturkonfiguration zugeordnet ist. Verwenden Sie dazu die in den vorherigen Themen beschriebenen Techniken. Durch die Verwendung einer Codesignatur bei der Bereitstellung von Lambda-Funktionen verhindern Sie, dass böswillige Akteure, die möglicherweise Anmeldeinformationen für die Erstellung oder Änderung von Funktionen erhalten haben, bösartigen Code in Ihre Funktionen einschleusen.

Automatisieren Sie Sicherheitsbewertungen für Lambda mit Amazon Inspector

Amazon Inspector ist ein automatisierter Schwachstellen-Management-Service, der Workloads kontinuierlich auf bekannte Softwareschwachstellen und unbeabsichtigte Netzwerkfreigabe durchsucht. Amazon Inspector erstellt einen Bericht, der die Schwachstelle beschreibt, die betroffene Ressource identifiziert, den Schweregrad der Schwachstelle bewertet und Hilfestellung zur Behebung gibt.

Amazon Inspector bietet eine kontinuierliche, automatisierte Schwachstellenanalyse für Lambda-Funktionen und -Layer. Amazon Inspector stellt zwei Scantypen für Lambda bereit:

- Lambda-Standardscan (Standard): Scannt Anwendungsabhängigkeiten innerhalb einer Lambda-Funktion und ihrer Layer auf Paketschwachstellen.
- Lambda-Codescan: Scannt den benutzerdefinierten Anwendungscode in Funktionen und Layern auf Code-Schwachstellen. Sie können entweder den Lambda-Standardscan einzeln oder zusammen mit dem Lambda-Codescan aktivieren.

Um Amazon Inspector zu aktivieren, navigieren Sie zur Amazon-Inspector-Konsole, erweitern Sie den Bereich Einstellungen und wählen Sie Kontoverwaltung. Wählen Sie auf der Registerkarte Konten die Option Aktivieren und wählen Sie dann eine der Scanoptionen aus.

Sie können Amazon Inspector für mehrere Konten aktivieren und bei der Einrichtung von Amazon Inspector die Berechtigungen zur Verwaltung von Amazon Inspector für die Organisation an bestimmte Konten delegieren. Während der Aktivierung müssen Sie Amazon Inspector Berechtigungen erteilen, indem Sie die folgende Rolle erstellen: `AWSServiceRoleForAmazonInspector2`. In der Amazon Inspector-Konsole können Sie diese Rolle mithilfe einer Option in einem Schritt erstellen.

Beim Lambda-Standardscan initiiert Amazon Inspector Schwachstellenscans von Lambda-Funktionen in den folgenden Situationen:

- Sobald Amazon Inspector eine bestehende Lambda-Funktion entdeckt.
- Wenn Sie eine neue Lambda-Funktion bereitstellen.
- Wenn Sie ein Update für den Anwendungscode oder die Abhängigkeiten einer vorhandenen Lambda-Funktion oder ihrer Layer bereitstellen.

- Immer wenn Amazon Inspector seiner Datenbank ein neues CVE-Element (Common Vulnerabilities and Exposures) hinzufügt und dieses CVE für Ihre Funktion relevant ist.

Beim Lambda-Codescan wertet Amazon Inspector Ihren Lambda-Funktionscode mithilfe von Automated Reasoning und Machine Learning aus. Dabei wird der Anwendungscode auf die allgemeine Einhaltung der Sicherheitsregeln hin analysiert. Wenn Amazon Inspector eine Sicherheitslücke in Ihrem Anwendungscode für Lambda-Funktionen entdeckt, erstellt Amazon Inspector einen detaillierten Schwachstellenbericht für den Code. Eine Liste möglicher Erkennungen finden Sie in der Amazon CodeGuru Detector-Bibliothek.

Rufen Sie die Amazon-Inspector-Konsole auf, um die Berichte einzusehen. Wählen Sie im Menü Funde die Option Nach Lambda-Funktion, um die Ergebnisse der Sicherheitsscans anzuzeigen, die für Lambda-Funktionen durchgeführt wurden.

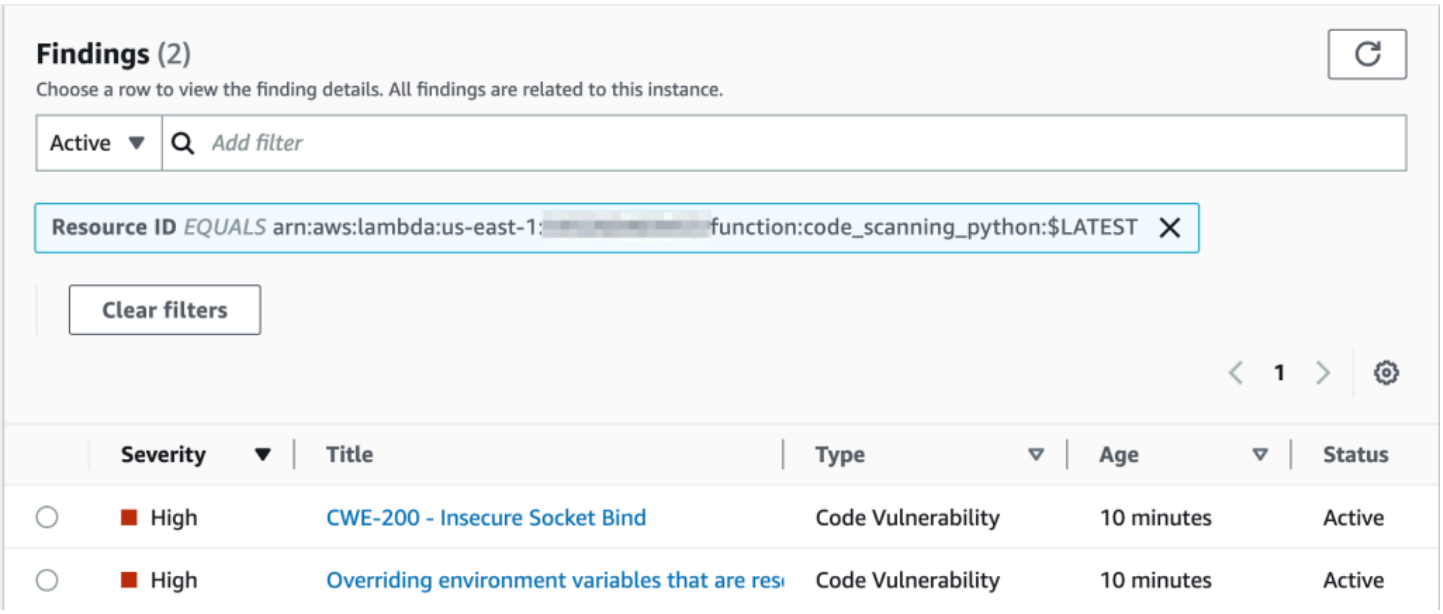
Um eine Lambda-Funktion vom Standardscan auszuschließen, kennzeichnen Sie die Funktion mit dem folgenden Schlüssel-Wert-Paar:

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

Um eine Lambda-Funktion von Codescans auszuschließen, kennzeichnen Sie die Funktion mit dem folgenden Schlüssel-Wert-Paar:

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

Wie in der folgenden Abbildung dargestellt, erkennt Amazon Inspector Schwachstellen automatisch und kategorisiert die Funde beispielsweise als Codeschwachstelle. Das deutet darauf hin, dass sich die Schwachstelle im Funktionscode befindet und nicht in einer der codeabhängigen Bibliotheken. Sie können diese Details für eine bestimmte Funktion oder mehrere Funktionen gleichzeitig einsehen.



Sie können jeden Befund eingehend untersuchen und erfahren, wie Sie das Problem beheben können.

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview	
AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details	
File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

Achten Sie bei der Verwendung von Lambda-Funktionen darauf, dass Sie die Namenskonventionen einhalten. Weitere Informationen finden Sie unter Arbeiten mit Lambda-Umgebungsvariablen.

Sie sind für die Lösungsvorschläge verantwortlich, die Sie akzeptieren. Lesen Sie sich die Lösungsvorschläge immer durch, bevor Sie sie annehmen. Möglicherweise müssen Sie Anpassungen an den Lösungsvorschlägen vornehmen, damit Ihr Code am Ende auch das tut, was Sie beabsichtigt haben.

Implementieren von Beobachtbarkeit für Lambda-Sicherheit und -Compliance

AWS Config ist ein nützliches Tool, um nicht konforme AWS serverlose Ressourcen zu finden und zu reparieren. Jede Änderung, die Sie an Ihren serverlosen Ressourcen vornehmen, wird in aufgezeichnet. AWS Config Darüber hinaus können Sie mit AWS Config Konfigurations-Snapshot-Daten auf S3 speichern.

Sie können Amazon Athena und Amazon Quick verwenden, um Dashboards zu erstellen und Daten anzuzeigen AWS Config . In Erkennen Sie nicht konforme Lambda-Bereitstellungen und -Konfigurationen mit AWS Config haben wir besprochen, wie sich bestimmte Konfigurationen wie Lambda-Layer visualisieren lassen. Dieses Thema erweitert diese Konzepte.

Abrufen von Informationen zu Lambda-Konfigurationen

Sie können Abfragen verwenden, um wichtige Konfigurationen wie AWS-Konto ID, Region, AWS X-Ray Tracing-Konfiguration, VPC-Konfiguration, Speichergröße, Laufzeit und Tags abzurufen. Hier ist eine Beispielabfrage, mit der Sie diese Informationen von Athena abrufen können:

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
```

```
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
  FROM
    unnested
```

Sie können die Abfrage verwenden, um ein Quick-Dashboard zu erstellen und die Daten zu visualisieren. Informationen zum Aggregieren von AWS Ressourcenkonfigurationsdaten, zum Erstellen von Tabellen in Athena und zum Erstellen von Quick-Dashboards mit den Daten aus Athena finden Sie unter Visualisieren von AWS Config Daten mit Athena und Amazon Quick im Blog AWS Cloud Operations and Management. Diese Abfrage ruft auch Tag-Informationen für die Funktionen ab. Dies ermöglicht eine genauere Untersuchung Ihrer Workloads und Umgebungen, insbesondere wenn Sie benutzerdefinierte Tags verwenden.

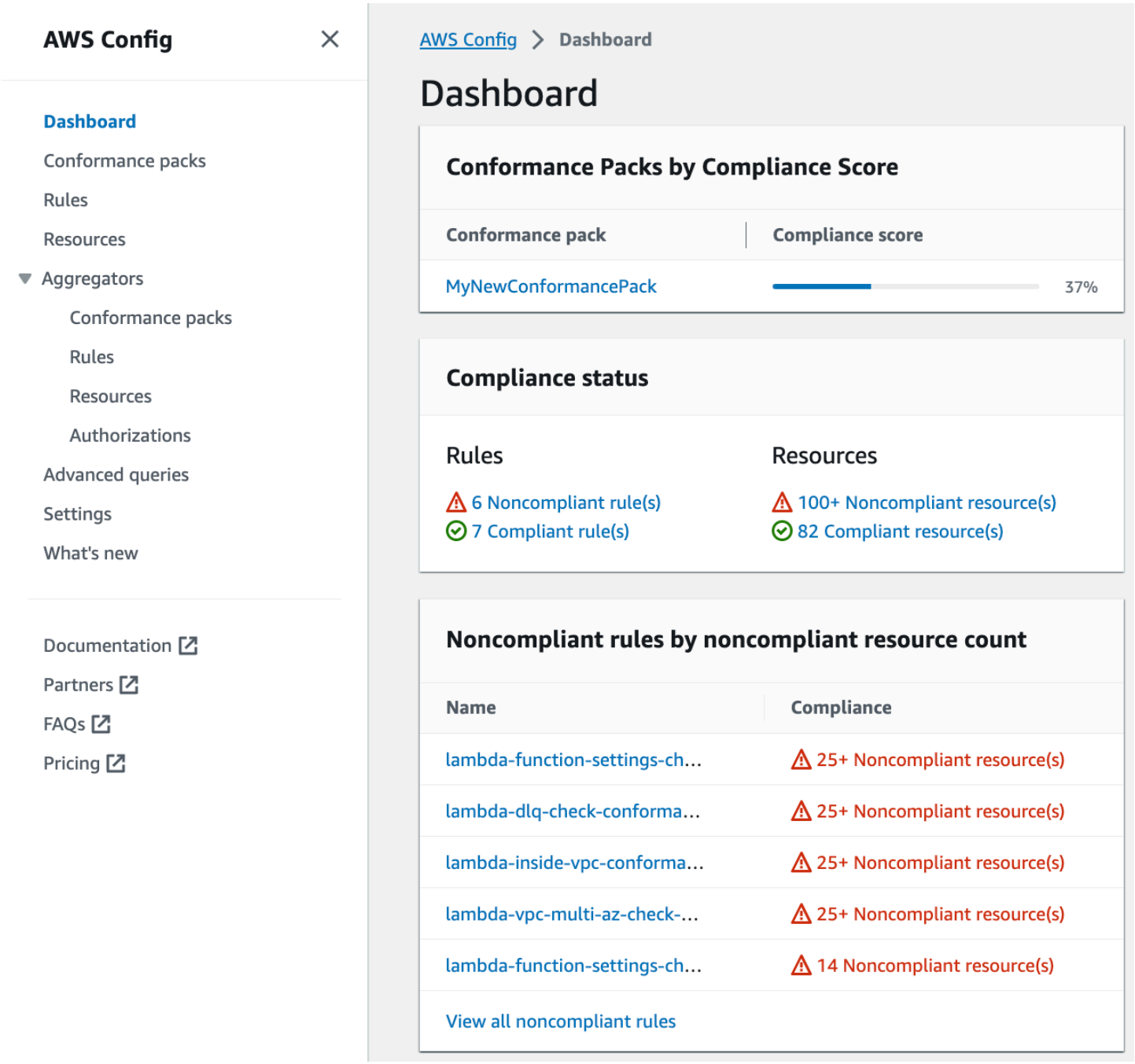


Weitere Informationen über die Aktionen, die Sie ergreifen können, finden Sie im Abschnitt Behandlung von Beobachtbarkeitsbefunden weiter unten in diesem Thema.

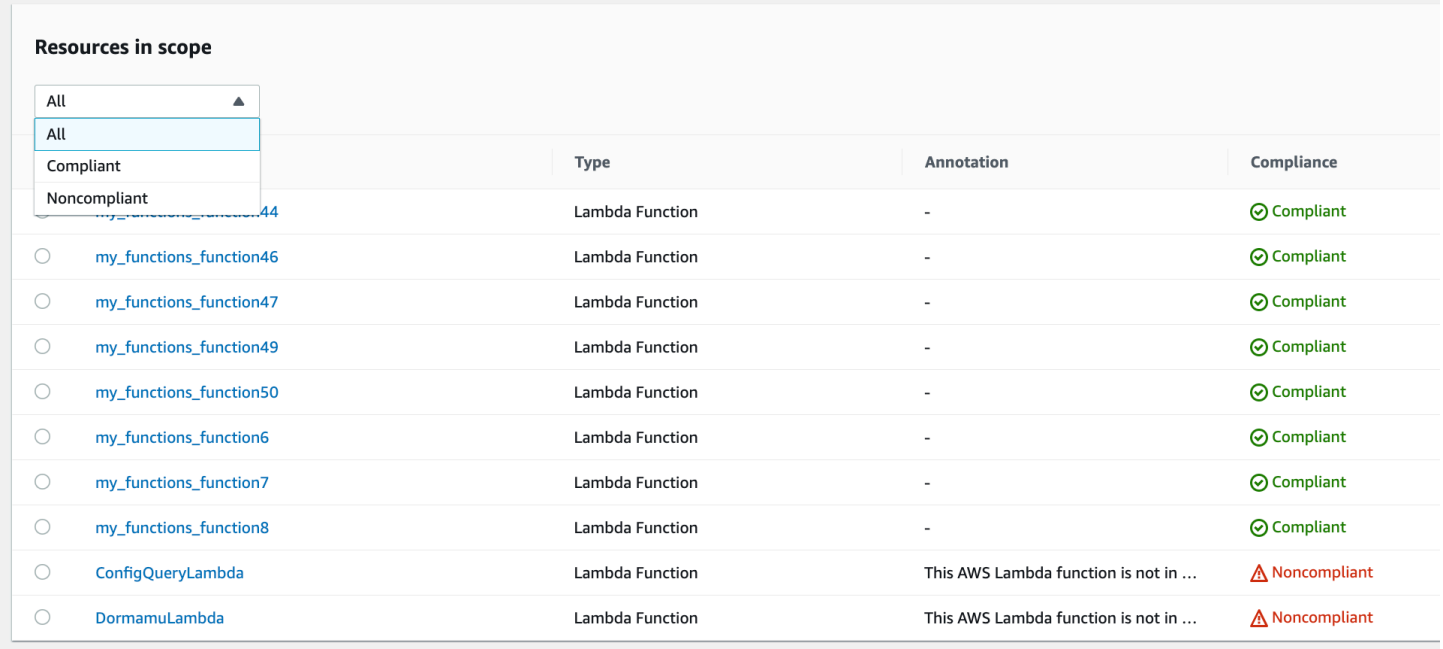
Abrufen von Informationen zur Lambda-Konformität

Mit den von generierten Daten können Sie Dashboards auf Organisationsebene erstellen AWS Config, um die Einhaltung der Vorschriften zu überwachen. Dies ermöglicht eine konsistente Nachverfolgung und Überwachung von:

- Compliance-Paketen nach Konformitätsbewertung
- Regeln von nicht konformen Ressourcen
- Compliance status (Compliance-Status)

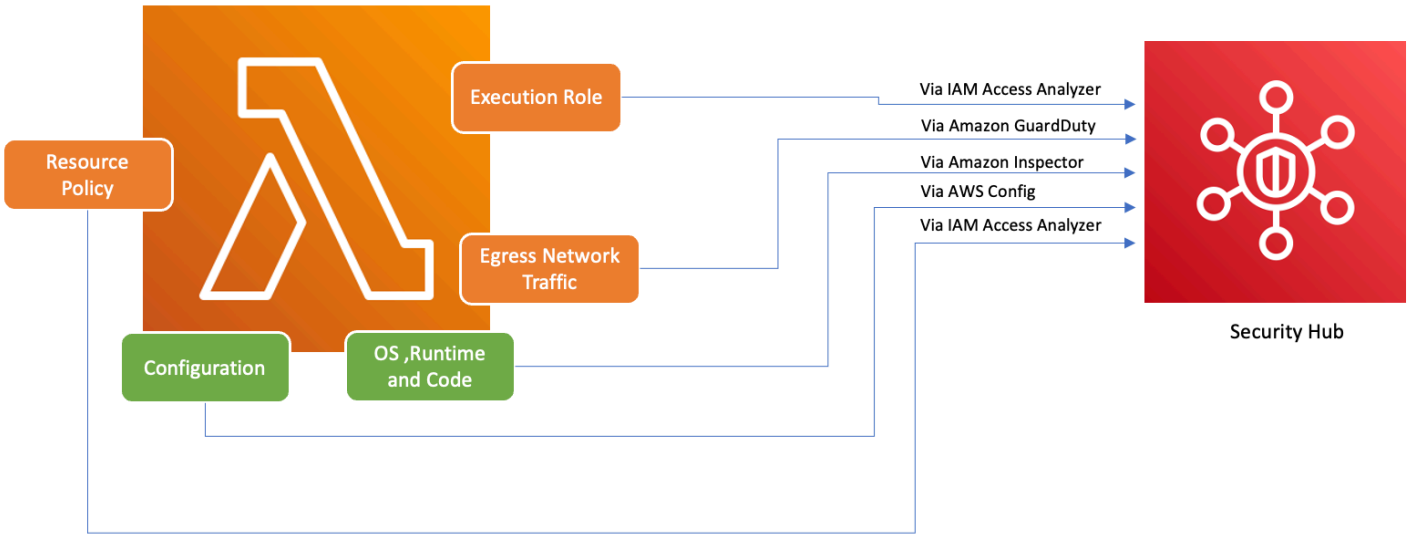


Überprüfen Sie jede Regel, um Ressourcen zu identifizieren, die für diese Regel nicht konform sind. Wenn Ihre Organisation beispielsweise vorschreibt, dass alle Lambda-Funktionen einer VPC zugeordnet werden müssen, und Sie eine AWS Config Regel zur Feststellung der Einhaltung der Vorschriften bereitgestellt haben, können Sie die Regel in der `lambda-inside-vpc` obigen Liste auswählen.



Weitere Informationen über die Aktionen, die Sie ergreifen können, finden Sie im Abschnitt [Behandlung von Beobachtbarkeitsbefunden] weiter unten.

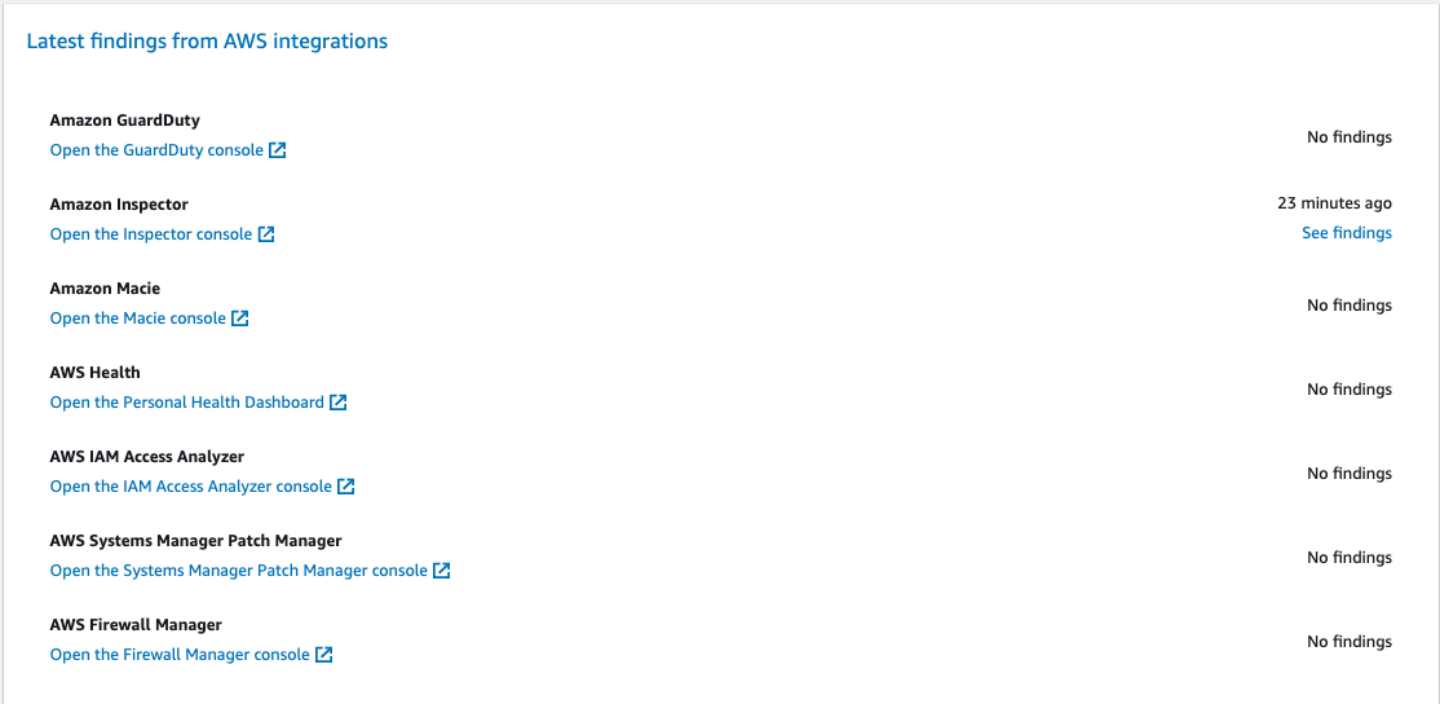
Einblick in die Grenzen der Lambda-Funktionen mithilfe von Security Hub CSPM



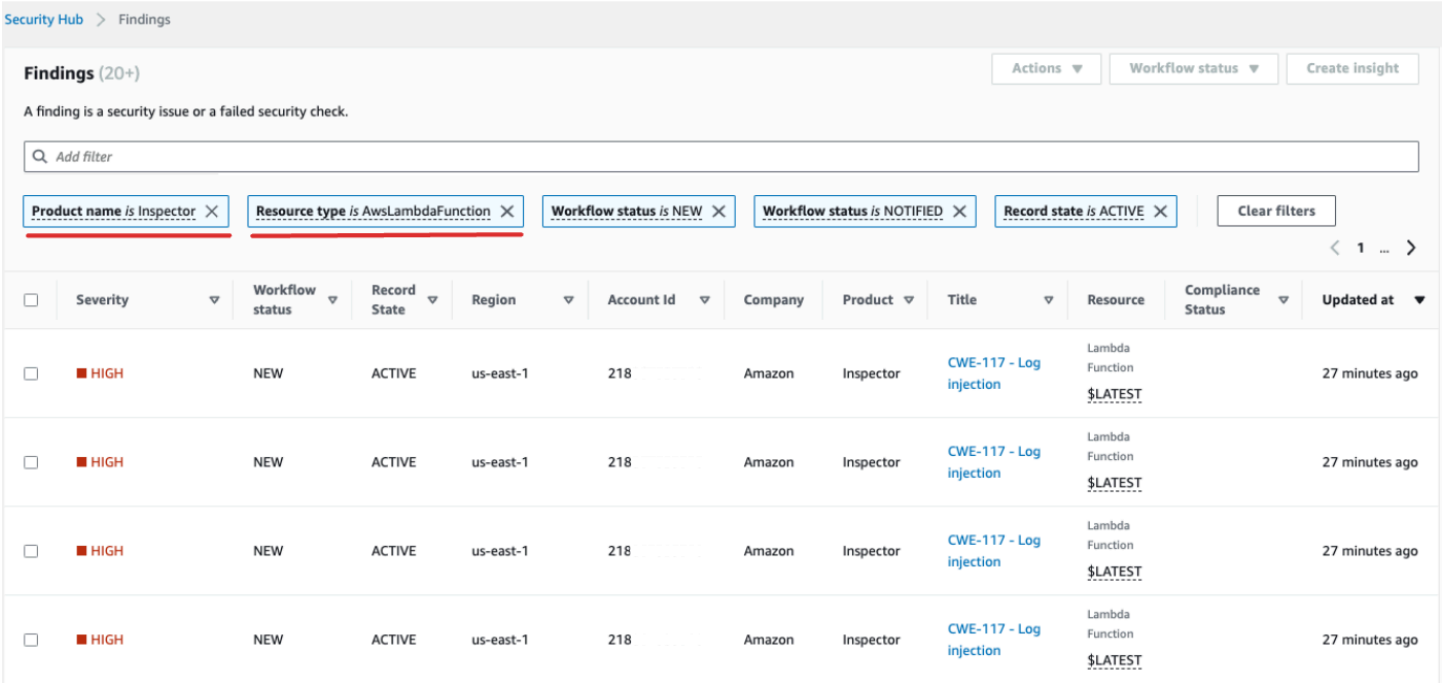
Um sicherzustellen, dass AWS Dienste wie Lambda sicher genutzt werden, AWS wurden die Foundational Security Best Practices v1.0.0 eingeführt. Diese Reihe von Best Practices enthält klare Richtlinien für den Schutz von Ressourcen und Daten in der AWS Umgebung und betont, wie wichtig es ist, ein hohes Sicherheitsniveau aufrechtzuerhalten. Das AWS Security Hub CSPM ergänzt dies durch ein einheitliches Sicherheits- und Compliance-Center. Es aggregiert, organisiert und priorisiert

Sicherheitserkenntnisse aus mehreren AWS Diensten wie Amazon Inspector und Amazon. AWS Identity and Access Management Access Analyzer GuardDuty

Wenn Sie Security Hub CSPM, Amazon Inspector und IAM Access Analyzer in Ihrem AWS Unternehmen aktiviert haben und GuardDuty diese aktiviert sind, aggregiert Security Hub CSPM automatisch die Ergebnisse dieser Dienste. Betrachten wir zum Beispiel Amazon Inspector. Mit Security Hub CSPM können Sie Code- und Paketschwachstellen in Lambda-Funktionen effizient identifizieren. Navigieren Sie in der Security Hub CSPM-Konsole zum unteren Abschnitt mit der Bezeichnung Aktuelle Erkenntnisse aus Integrationen. AWS Hier können Sie Ergebnisse aus verschiedenen integrierten Diensten einsehen und analysieren. AWS



Für Details klicken Sie in der zweiten Spalte auf den Link Befunde anzeigen. Daraufhin wird eine nach Produkten gefilterte Liste mit Ergebnissen angezeigt, z. B. Amazon Inspector. Um Ihre Suche auf Lambda-Funktionen zu beschränken, setzen Sie `ResourceType` auf `AwsLambdaFunction`. Nun werden Ergebnisse von Amazon Inspector zu Lambda-Funktionen angezeigt.



Denn GuardDuty Sie können verdächtige Muster im Netzwerkverkehr identifizieren. Solche Anomalien könnten auf die Existenz von potenziell bösartigem Code in Ihrer Lambda-Funktion hindeuten.

Mit IAM Access Analyzer können Sie Richtlinien überprüfen, insbesondere solche mit Bedingungsanweisungen, die externen Entitäten Funktionszugriff gewähren. Darüber hinaus wertet IAM Access Analyzer die Berechtigungen aus, die festgelegt wurden, wenn der AddPermission Vorgang in der Lambda-API zusammen mit einem verwendet wird. `EventSourceToken`

Behandlung von Beobachtbarkeitsbefunden

Angesichts der vielfältigen Konfigurationen, die für Lambda-Funktionen möglich sind, und ihrer unterschiedlichen Anforderungen ist eine standardisierte Automatisierungslösung für die Problembehebung möglicherweise nicht für jede Situation geeignet. Darüber hinaus werden Änderungen in den verschiedenen Umgebungen unterschiedlich implementiert. Wenn Sie auf eine Konfiguration stoßen, die nicht konform zu sein scheint, beachten Sie die folgenden Richtlinien:

1. Strategie zur Kennzeichnung

   Wir empfehlen die Implementierung einer umfassenden Kennzeichnungsstrategie. Jede Lambda-Funktion sollte mit folgenden Schlüsselinformationen gekennzeichnet sein:

   - Eigentümer: Die Person oder das Team, die für die Funktion verantwortlich ist.

- Umgebung: Produktion, Staging, Entwicklung oder Sandbox.
- Anwendung: Der breitere Kontext, zu dem diese Funktion gehört, falls zutreffend.

2. Kontakt zum Eigentümer

   Anstatt notwendige Änderungen zu automatisieren (wie die Anpassung der VPC-Konfiguration), sollten Sie sich an die Eigentümer nicht konformer Funktionen (siehe Eigentümer-Tag) wenden und ihnen genügend Zeit geben, um entweder:

   - nicht konforme Konfigurationen für Lambda-Funktionen anzupassen
   - eine Erklärung bereitzustellen und eine Ausnahme anzufordern oder die Compliance-Standards anzupassen

3. Fühen Sie eine Configuration Management Database (CMDB)

   Tags können zwar unmittelbar Kontext liefern, eine zentralisierte CMDB kann jedoch genauere Informationen liefern. Die Datenbank kann detaillierte Informationen über jede Lambda-Funktion, ihre Abhängigkeiten und andere wichtige Metadaten enthalten. Eine CMDB ist eine unschätzbare Ressource für Audits, Konformitätsprüfungen und die Identifizierung der Eigentümer einer Funktion.

Da sich Serverless-Infrastrukturen kontinuierlich weiterentwickeln, ist es wichtig, Überwachungsmaßnahmen proaktiv zu organisieren. Mit Tools wie AWS Config Security Hub CSPM und Amazon Inspector können potenzielle Anomalien oder nicht konforme Konfigurationen schnell identifiziert werden. Tools allein können jedoch keine vollständige Konformität oder optimale Konfigurationen gewährleisten. Es ist wichtig, diese Tools mit gut dokumentierten Prozessen und bewährten Verfahren zu kombinieren.

- Feedback-Schleife: Sobald Korrekturmaßnahmen ergriffen werden, sollten Sie sicherstellen, dass es eine Feedback-Schleife gibt. Dabei werden nicht konforme Ressourcen regelmäßig überprüft, um zu ermitteln, ob sie aktualisiert wurden oder immer noch dieselben Probleme haben.
- Dokumentation: Beobachtungen, Maßnahmen und alle gewährten Ausnahmen sollten immer dokumentiert werden. Eine ordnungsgemäße Dokumentation hilft nicht nur bei Audits, sondern trägt auch dazu bei, den Prozess zu verbessern, um die Compliance und Sicherheit dauerhaft zu stärken.
- Schulung und Sensibilisierung: Alle Beteiligten, insbesondere die Eigentümer der Lambda-Funktionen, sollten regelmäßig geschult und über bewährte Verfahren, Unternehmensrichtlinien und Compliance-Vorschriften informiert werden. Regelmäßige Workshops, Webinare oder

Schulungen können einen großen Beitrag dazu leisten, dass alle Beteiligten bezüglich Sicherheit und Compliance auf dem gleichen Stand sind.

Zusammenfassend lässt sich sagen, dass Tools und Technologien zwar die Erkennung und Meldung von potenziellen Problemen unterstützen, der Faktor Mensch –Verständnis, Kommunikation, Schulung und Dokumentation – aber weiterhin von zentraler Bedeutung ist. Beides zusammen ergibt eine leistungsstarke Kombination, um sicherzustellen, dass Ihre Lambda-Funktionen und die größere Infrastruktur konform, sicher und für Ihre Geschäftsanforderungen optimiert sind.

Konformitätsüberprüfung für AWS Lambda

Third-party Auditoren bewerten die Sicherheit und Compliance von im AWS Lambda Rahmen mehrerer AWS Compliance-Programme. Hierzu zählen unter anderem SOC, PCI, FedRAMP und HIPAA.

Eine Liste der AWS Dienstleistungen, die Gegenstand bestimmter Compliance-Programme sind, finden Sie unter AWS Dienstleistungen im Umfang nach Compliance-Programmen. Allgemeine Informationen finden Sie unter AWS -Compliance-Programme.

Auditberichte von Drittanbietern können Sie mit herunterladen AWS Artifact. Weitere Informationen finden Sie unter Herunterladen von Berichten in AWS -Artifact.

Ihre Compliance-Verantwortung bei Verwendung von Lambda hängt von der Vertraulichkeit der Daten, den Compliance-Zielen des Unternehmens und den geltenden Gesetzen und Vorschriften ab. Sie können Governance-Kontrollen implementieren, um sicherzustellen, dass die Lambda-Funktionen Ihres Unternehmens Ihren Compliance-Anforderungen entsprechen. Weitere Informationen finden Sie unter Erstellen Sie eine Governance-Strategie für Lambda-Funktionen und -Layer.

Informationen zum AWS Schutz der globalen Infrastruktur, auf der Lambda ausgeführt wird, finden Sie unterthe section called “Sicherheit der Infrastruktur”.

Resilienz in AWS Lambda

Die AWS globale Infrastruktur basiert auf AWS Regionen und Availability Zones. AWS Regionen bieten mehrere physisch getrennte und isolierte Availability Zones, die über Netzwerke mit niedriger Latenz, hohem Durchsatz und hochredundanten Netzwerken miteinander verbunden sind. Mithilfe von Availability Zones können Sie Anwendungen und Datenbanken erstellen und ausführen, die automatisch Failover zwischen Availability Zones ausführen, ohne dass es zu Unterbrechungen kommt. Availability Zones sind besser hoch verfügbar, fehlertoleranter und skalierbarer als herkömmliche Infrastrukturen mit einem oder mehreren Rechenzentren.

Weitere Informationen zu AWS Regionen und Availability Zones finden Sie unter AWS Globale Infrastruktur.

Zusätzlich zur AWS globalen Infrastruktur bietet Lambda mehrere Funktionen, mit denen Sie Ihre Anforderungen an Datenstabilität und Datensicherung erfüllen können.

- Versioning – Sie können in Lambda Versioning verwenden, um Code und Konfiguration Ihrer Funktion während ihrer Entwicklung zu speichern. Zusammen mit Aliasnamen können Sie die

Versionierung verwenden, um Bereitstellungen durchzuführen und fortlaufend durchzuführen blue/green. Details hierzu finden Sie unter Versionen der Lambda-Ereignisfunktion verwalten.

- Skalierung – Wenn Ihre Funktion während der Verarbeitung einer früheren Anforderung eine weitere Anforderung empfängt, startet Lambda eine weitere Instance Ihrer Funktion, um die höhere Last zu verarbeiten. Lambda wird automatisch skaliert, um 1.000 gleichzeitige Ausführungen pro Region zu verarbeiten, ein Kontingent, das bei Bedarf erhöht werden kann. Details hierzu finden Sie unter Die Skalierung von Lambda-Ereignisfunktionen verstehen.
- Hohe Verfügbarkeit – Lambda führt Ihre Funktion in mehreren Availability Zones aus, um sicherzustellen, dass sie für die Verarbeitung von Ereignissen verfügbar ist, wenn der Service in einer Zone unterbrochen wird. Wenn Sie Ihre Funktion für die Herstellung von Verbindungen mit einer Virtual Private Cloud (VPC) in Ihrem Konto konfigurieren, geben Sie Subnetze in mehreren Availability Zones an, um eine hohe Verfügbarkeit sicherzustellen. Details hierzu finden Sie unter Lambda Event Functions Zugriff auf Ressourcen in einer Amazon VPC gewähren.
- Reservierte Gleichzeitigkeit – Um sicherzustellen, dass Ihre Funktion stets skaliert werden kann, um zusätzliche Anforderungen zu verarbeiten, können Sie für sie Gleichzeitigkeit reservieren. Die Reservierung von Gleichzeitigkeit für eine Funktion stellt sicher, dass sie bis zu einer angegebenen Anzahl gleichzeitiger Aufrufe skaliert werden kann, diese Anzahl jedoch nicht überschreitet. Auf diese Weise wird sichergestellt, dass keine Anforderungen verloren gehen, da andere Funktionen der gesamte verfügbare Gleichzeitigkeit verbrauchen. Details hierzu finden Sie unter Konfiguration reservierter Parallelität für eine Event-Funktion.
- Wiederholversuche – Im Fall asynchroner Aufrufe und bestimmter, von anderen Diensten ausgelöster Aufrufe führt Lambda bei Fehlern automatisch Wiederholversuche mit Verzögerungen zwischen den einzelnen Wiederholversuchen aus. Für die Ausführung der Wiederholversuche sind andere Clients und AWS-Services , die Funktionen synchron aufrufen, verantwortlich. Details hierzu finden Sie unter Grundlegendes zum Wiederholungsverhalten in Lambda-Ereignisfunktionen.
- Dead-letter Warteschlange — Für asynchrone Aufrufe können Sie Lambda so konfigurieren, dass Anfragen an eine Warteschlange mit unlesbaren Buchstaben gesendet werden, wenn alle Wiederholungsversuche fehlschlagen. Eine Warteschlange für unzustellbare Nachrichten ist ein Amazon-SNS-Thema oder eine Amazon-SQS-Warteschlange, die Ereignisse zur Fehlerbehebung oder erneuten Verarbeitung empfängt. Details hierzu finden Sie unter Hinzufügen einer Warteschlange für unzustellbare Nachrichten.

Weitere Informationen über das Netzwerk und die physische Infrastruktur, die Lambda schützt, finden Sie unter. the section called “Sicherheit der Infrastruktur”

Sicherheit der Infrastruktur in AWS Lambda

Als verwalteter Service ist AWS Lambda durch die globalen Verfahren zur Gewährleistung der Netzwerksicherheit von AWS geschützt. Informationen zu AWS-Sicherheitsdiensten und wie AWS die Infrastruktur schützt, finden Sie unter AWSCloud-Sicherheit. Informationen zum Entwerfen Ihrer AWS-Umgebung anhand der bewährten Methoden für die Infrastruktursicherheit finden Sie unter Infrastrukturschutz im Security Pillar AWS Well-Architected Framework.

Sie verwenden durch AWS veröffentlichte API-Aufrufe, um über das Netzwerk auf Lambda zuzugreifen. Kunden müssen Folgendes unterstützen:

- Transport Layer Security (TLS). Wir benötigen TLS 1.2 und empfehlen TLS 1.3.
- Verschlüsselungs-Suiten mit Perfect Forward Secrecy (PFS) wie DHE (Ephemeral Diffie-Hellman) oder ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Die meisten modernen Systeme wie Java 7 und höher unterstützen diese Modi.

Sicherung von Workloads mit öffentlichen Endpunkten

AWS Bietet für Workloads, die öffentlich zugänglich sind, eine Reihe von Funktionen und Diensten, die zur Minderung bestimmter Risiken beitragen können. Dieser Abschnitt behandelt die Authentifizierung und Autorisierung von Anwendungsbenutzern und den Schutz von API-Endpunkten.

Authentifizierung und Autorisierung

Authentifizierung bezieht sich auf Identität und Autorisierung bezieht sich auf Aktionen. Verwenden Sie die Authentifizierung, um zu kontrollieren, wer eine Lambda-Funktion aufrufen kann und verwenden Sie dann die Autorisierung, um zu kontrollieren, was sie tun können. Für viele Anwendungen ist IAM ausreichend, um beide Kontrollmechanismen zu verwalten.

Für Anwendungen mit externen Benutzern, wie z. B. Web- oder Mobilanwendungen, ist es üblich, JSON-Web-Tokens (JWTs) zur Verwaltung der Authentifizierung und Autorisierung zu verwenden. Im Gegensatz zur herkömmlichen, serverbasierten Passwortverwaltung werden JWTs bei jeder Anfrage vom Client übergeben. Sie sind eine kryptografisch sichere Methode, um Identität und Ansprüche anhand der vom Client übermittelten Daten zu überprüfen. Bei Lambda-based Anwendungen können Sie jeden Aufruf an jedem API-Endpunkt sichern, ohne sich bei der Authentifizierung auf einen zentralen Server verlassen zu müssen.

Sie können JWTs mit Amazon Cognito implementieren, einem Benutzerverzeichnisdienst, der Registrierung, Authentifizierung, Kontowiederherstellung und andere gängige Kontoverwaltungsvorgänge abwickeln kann. Amplify Framework bietet Bibliotheken, die die Integration dieses Dienstes in Ihre Frontend-Anwendung vereinfachen. Sie können auch Partnerdienste von Drittanbietern wie Auth0 erwägen.

Angesichts der wichtigen Sicherheitsrolle eines Identitätsanbieter-Dienstes ist es wichtig, professionelle Tools zum Schutz Ihrer Anwendung zu verwenden. Es wird nicht empfohlen, eigene Dienste für die Authentifizierung oder Autorisierung zu entwickeln. Alle Sicherheitslücken in benutzerdefinierten Bibliotheken können erhebliche Auswirkungen auf die Sicherheit Ihres Workloads und seiner Daten haben.

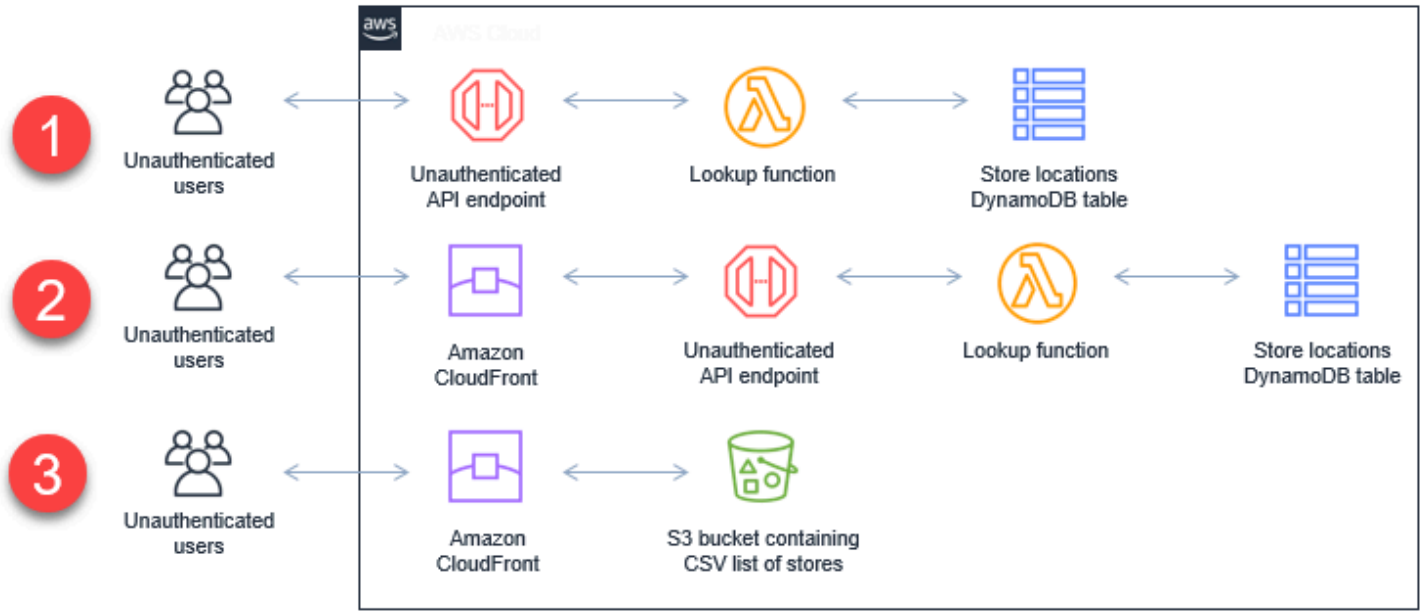
API-Endpunkte schützen

Für Serverless-Anwendungen ist die Verwendung von Amazon API Gateway die bevorzugte Methode, eine Backend-Anwendung öffentlich bereitzustellen. Dies kann Ihnen helfen, eine API vor böswilligen Benutzern oder Datenverkehrsspitzen zu schützen.

API Gateway bietet zwei Endpunkttypen für Serverless-Entwickler: REST-APIs und HTTP-APIs. Beide unterstützen die Autorisierung mithilfe von AWS Lambda IAM oder Amazon Cognito. Bei der Verwendung von IAM oder Amazon Cognito werden eingehende Anfragen ausgewertet. Fehlt ein erforderliches Token oder enthält es eine ungültige Authentifizierung, wird die Anfrage abgelehnt. Diese Anfragen werden Ihnen nicht in Rechnung gestellt und sie werden auch nicht auf die Drosselungskontingente angerechnet.

Auf nicht authentifizierte API-Routen kann jeder im öffentlichen Internet zugreifen. Es wird daher empfohlen, die Verwendung nicht authentifizierter APIs einzuschränken. Wenn Sie nicht authentifizierte APIs verwenden müssen, ist es wichtig, diese vor häufigen Risiken wie Denial-of-Service (DoS) -Angriffen zu schützen. Die Anwendung von AWS WAF auf diese APIs kann dazu beitragen, Ihre Anwendung vor SQL-Injektion- und Cross-Site-Scripting (XSS)-Angriffen zu schützen. API Gateway implementiert auch Drosselung auf AWS Kontoebene und pro Client, wenn API-Schlüssel verwendet werden.

In vielen Fällen kann die Funktionalität einer nicht authentifizierten API mit einem alternativen Ansatz erreicht werden. Beispielsweise könnte eine Webanwendung Benutzern, die nicht angemeldet sind, eine Liste der Einzelhandelsgeschäfte von Kunden aus einer DynamoDB-Tabelle bereitstellen. Diese Anforderung kann von einer Frontend-Webanwendung oder von einer anderen Quelle stammen, die den URL-Endpunkt aufruft. In diesem Diagramm werden drei Lösungen verglichen:



1. Diese nicht authentifizierte API kann von jedem Benutzer im Internet aufgerufen werden. Bei einem Denial-of-Service-Angriff ist es möglich, die API-Drosselungsgrenzen, die Lambda-Parallelität oder die von DynamoDB bereitgestellte Lesekapazität in einer zugrunde liegenden Tabelle auszuschöpfen.
2. Eine CloudFront Verteilung vor dem API-Endpunkt mit einer geeigneten Time-to-Live-Konfiguration (TTL) würde bei einem DoS-Angriff den größten Teil des Datenverkehrs absorbieren, ohne die zugrunde liegende Lösung für das Abrufen der Daten zu ändern.
3. Bei statischen Daten, die sich selten ändern, könnte die CloudFront Distribution alternativ die Daten aus einem Amazon S3-Bucket bereitstellen.

Verwendung von Codesignatur zur Überprüfung der Codeintegrität mit Lambda

Die Codesignatur hilft sicherzustellen, dass nur vertrauenswürdiger Code in Ihren Lambda-Funktionen bereitgestellt wird. Mithilfe AWS Signer können Sie digital signierte Codepakete für Ihre Funktionen erstellen. Wenn Sie einer Funktion eine Code-Signaturkonfiguration hinzufügen, überprüft Lambda, ob alle neuen Codebereitstellungen von einer vertrauenswürdigen Quelle signiert sind. Da die Validierungsprüfungen von Codesignatur zum Zeitpunkt der Bereitstellung ausgeführt werden, gibt es keine Auswirkungen auf die Ausführung der Funktion. Informationen zur Verwaltung von Codesignaturrichtlinien finden Sie unterthe section called “Berechtigungen”.

> **Note**
>
> Die Codesignatur gilt für Lambda-Ereignisfunktionen, die als .zip- oder Container-Image-Pakete bereitgestellt werden. Es wird nicht von Lambda Web Functions oder Lambda MicroVMS verwendet.

> **Important**
>
> Codesignatur-Konfigurationen verhindern nur neue Bereitstellungen von unsigniertem Code. Wenn Sie einer vorhandenen Funktion mit unsigniertem Code eine Codesignatur-Konfiguration hinzufügen, wird dieser Code so lange ausgeführt, bis Sie ein neues Codepaket bereitstellen.

Wenn Sie die Codesignatur für eine Funktion aktivieren, müssen alle Ebenen, die Sie der Funktion hinzufügen, auch von einem zulässigen Signaturprofil signiert werden.

Es fallen keine zusätzlichen Gebühren für die Nutzung AWS Signer oder das Signieren von Code an. AWS Lambda

Signaturvalidierung

Lambda führt die folgenden Validierungsprüfungen durch, wenn Sie ein signiertes Codepaket für Ihre Funktion bereitstellen:

1. Integrität: Überprüft, dass das Codepaket seit seiner Unterzeichnung nicht geändert wurde. Lambda vergleicht den Hash des Pakets mit dem Hash aus der Signatur.
2. Ablauf: Bestätigt, dass die Signatur des Codepakets nicht abgelaufen ist.
3. Diskrepanz: Bestätigt, dass das Codepaket mit einem der zulässigen Signaturprofil signiert ist.
4. Widerruf: Bestätigt, dass die Signatur des Codepakets nicht widerrufen wurde.

Wenn Sie eine Codesignaturkonfiguration erstellen, können Sie den UntrustedArtifactOnDeploymentParameter verwenden, um anzugeben, wie Lambda reagieren soll, wenn die Ablauf-, Nichtübereinstimmungs- oder Sperrprüfungen fehlschlagen. Sie können eine der folgenden Aktionen auswählen:

- `Warn`: Dies ist die Standardeinstellung. Lambda ermöglicht die Bereitstellung des Codepakets, gibt jedoch eine Warnung aus. Lambda gibt eine neue CloudWatch Amazon-Metrik (`SignatureValidationErrors`) aus und speichert die Warnung auch im CloudTrail Protokoll.
- `Enforce` – Lambda gibt eine Warnung aus (wie bei der `Warn`-Aktion) und blockiert die Bereitstellung des Codepakets.

Themen

- Erstellen von Codesignatur-Konfigurationen für Lambda
- Konfigurieren von IAM-Richtlinien für Lambda-Codesignaturkonfigurationen
- Verwendung von Tags in Codesignatur-Konfigurationen

Erstellen von Codesignatur-Konfigurationen für Lambda

Um die Codesignatur für eine Funktion zu aktivieren, erstellen Sie eine Codesignatur-Konfiguration und hängen sie an die Funktion an. Eine Codesignatur-Konfiguration definiert eine Liste der zulässigen Signaturprofile und die Richtlinienaktion, die durchgeführt werden soll, wenn eine der Validierungsprüfungen fehlschlägt.

> **Note**
>
> Funktionen, die als Container-Images definiert sind, unterstützen keine Codesignatur.

Sections

- Voraussetzungen für die Konfiguration
- Erstellen von Codesignatur-Konfigurationen
- Aktivieren der Codesignatur für eine Funktion

Voraussetzungen für die Konfiguration

Bevor Sie die Codesignatur für eine Lambda-Funktion konfigurieren können, verwenden Sie AWS Signer, um Folgendes zu tun:

- Erstellen Sie mindestens ein Signaturprofil.
- Verwenden Sie ein Signaturprofil, um ein signiertes Codepaket für Ihre Funktion zu erstellen.

Erstellen von Codesignatur-Konfigurationen

Mit einer Codesignatur-Konfiguration wird eine Liste der zulässigen Signaturprofile und die Richtlinie für Signaturvalidierung definiert.

So erstellen Sie eine Codesignatur-Konfiguration (Konsole)

1. Öffnen Sie die Seite Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie Create configuration (Konfiguration erstellen).
3. Geben Sie unter Description (Beschreibung) einen aussagekräftigen Namen für die Konfiguration ein.
4. Fügen Sie der Konfiguration unter Signaturprofile bis zu 20 Signaturprofile hinzu.
   a. Wählen Sie für ARN der Signaturprofilversion den Amazon-Ressourcennamen (ARN) einer Profilversion aus oder geben Sie den ARN ein.
   b. Um ein zusätzliches Signaturprofil hinzuzufügen, wählen Sie Signaturprofil hinzufügen aus.
5. Wählen Sie unter Richtlinie zur Signaturvalidierung die Option Warnen oder Durchsetzen aus.
6. Wählen Sie Create configuration (Konfiguration erstellen).

Aktivieren der Codesignatur für eine Funktion

Um die Codesignatur für eine Funktion zu aktivieren, fügen Sie der Funktion eine Codesignatur-Konfiguration hinzu.

> **Important**
>
> Codesignatur-Konfigurationen verhindern nur neue Bereitstellungen von unsigniertem Code. Wenn Sie einer vorhandenen Funktion mit unsigniertem Code eine Codesignatur-Konfiguration hinzufügen, wird dieser Code so lange ausgeführt, bis Sie ein neues Codepaket bereitstellen.

So verknüpfen Sie eine Codesignatur-Konfiguration mit einer Funktion (Konsole)

1. Öffnen Sie die Seite Event Functions der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie die Codesignatur aktivieren möchten.

3. Öffnen Sie die Registerkarte Configuration (Konfiguration).
4. Scrollen Sie nach unten und wählen Sie Codesignatur aus.
5. Wählen Sie Bearbeiten aus.
6. Wählen Sie i n Codesignatur bearbeiten eine Codesignatur-Konfiguration für diese Funktion aus.
7. Wählen Sie Speichern.

Konfigurieren von IAM-Richtlinien für Lambda-Codesignaturkonfigurationen

Um einem Benutzer die Berechtigung zum Zugriff auf die Lambda-Codesignatur-API-Vorgänge zu erteilen, fügen Sie der Benutzerrichtlinie eine oder mehrere Richtlinienanweisungen bei. Weitere Informationen zu Benutzerrichtlinien finden Sie unter Identity-based IAM-Richtlinien für Lambda.

Die folgende Beispiel-Richtlinienanweisung gewährt die Berechtigung zum Erstellen, Aktualisieren und Abrufen von Codesignatur-Konfigurationen.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

Administratoren können den `CodeSigningConfigArn`-Bedingungsschlüssel verwenden, um die Codesignatur-Konfigurationen anzugeben, mit denen Entwickler Ihre Funktionen erstellen oder aktualisieren müssen.

Die folgende Beispielrichtlinienanweisung gewährt die Berechtigung zum Erstellen einer Funktion. Die Richtlinienanweisung enthält eine `lambda:CodeSigningConfigArn`-Bedingung, um die

zulässige Konfiguration für die Codesignierung anzugeben. Lambda blockiert `CreateFunction` API-Anfragen, wenn der CodeSigningConfigArnParameter fehlt oder nicht mit dem Wert in der Bedingung übereinstimmt.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "lambda:CodeSigningConfigArn": "arn:aws:lambda:us-
east-1:111122223333:code-signing-config:csc-0d4518bd353a0a7c6"
        }
      }
    }
  ]
}
```

Verwendung von Tags in Codesignatur-Konfigurationen

Sie können Codesignaturkonfigurationen taggen, um Ihre Ressourcen zu organisieren und zu verwalten. Tags sind frei formulierte Schlüssel-Wert-Paare, die mit Ihren Ressourcen verknüpft sind und von AWS-Services unterstützt werden. Weitere Informationen zu Anwendungsfällen für Tags finden Sie unter Allgemeine Tagging-Strategien im Tagging AWS Resources and Tag Editor Guide.

Sie können die AWS Lambda API verwenden, um Tags anzuzeigen und zu aktualisieren. Sie können auch Tags anzeigen und aktualisieren, während Sie eine bestimmte Codesignatur-Konfiguration in der Lambda-Konsole verwalten.

Sections

- Erforderliche Berechtigungen zum Arbeiten mit Tags

- Verwendung von Tags mit der Lambda-Konsole
- Verwenden von Tags mit dem AWS CLI

Erforderliche Berechtigungen zum Arbeiten mit Tags

Um einer AWS Identity and Access Management -(IAM)-Identität (Benutzer, Gruppe oder Rolle) das Lesen oder Setzen von Tags auf einer Ressource zu ermöglichen, gewähren Sie ihr die entsprechenden Berechtigungen:

- lambda: ListTags —Wenn eine Ressource Tags enthält, gewähren Sie diese Berechtigung jedem, der sie aufrufen `ListTags` muss. Für getaggte Event-Funktionen ist diese Berechtigung auch für erforderlich. `GetFunction`
- lambda: TagResource —Erteilen Sie diese Berechtigung jedem, der bei create aufrufen `TagResource` oder ein Tag ausführen muss.

Optional können Sie erwägen, auch die lambda: UntagResource -Berechtigung zu erteilen, um `UntagResource` Aufrufe der Ressource zuzulassen.

Weitere Informationen finden Sie unter Identity-based IAM-Richtlinien für Lambda.

Verwendung von Tags mit der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um Codesignatur-Konfigurationen mit Tags zu erstellen, Tags zu vorhandenen Codesignatur-Konfigurationen hinzuzufügen und Codesignatur-Konfigurationen nach Tags zu filtern.

So fügen Sie ein Tag hinzu, wenn Sie eine Codesignatur-Konfiguration erstellen

1. Öffnen Sie Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie in der Kopfzeile des Inhaltsfensters die Option Konfiguration erstellen aus.
3. Richten Sie im Abschnitt oben im Inhaltsbereich Ihre Codesignatur-Konfiguration ein. Weitere Informationen zum Konfigurieren von Codesignatur-Konfigurationen finden Sie unter the section called “Codesignatur”.
4. Klicken Sie im Abschnitt Tags auf Neuen Tag hinzufügen.
5. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Tagging-Einschränkungen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Tagging AWS Resources and Tag Editor Guide.

6. Wählen Sie Create configuration (Konfiguration erstellen).

So fügen Sie einer vorhandenen Codesignaturkonfiguration ein Tag hinzu

1. Öffnen Sie Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie den Namen der Codesignatur-Konfiguration aus.
3. Wählen Sie auf den Registerkarten unter dem Detailbereich die Option Tags aus.
4. Wählen Sie Tags verwalten aus.
5. Wählen Sie Neues Tag hinzufügen aus.
6. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Einschränkungen beim Taggen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Leitfaden für AWS Tag-Ressourcen und Tag-Editor.
7. Wählen Sie Speichern.

So filtern Sie Codesignatur-Konfigurationen nach Tags

1. Öffnen Sie Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie das Suchfeld.
3. Wählen Sie in der Dropdown-Liste Ihr Tag unter der Tag-Unterüberschrift aus.
4. Wählen Sie Verwenden: „Tag-Name“, um alle Codesignaturkonfigurationen zu sehen, die mit dieser Taste gekennzeichnet sind oder wählen Sie einen Operator, um weiter nach Werten zu filtern.
5. Wählen Sie Ihren Tag-Wert aus, um nach einer Kombination aus Tag-Schlüssel und -Wert zu filtern.

Das Suchfeld unterstützt auch die Suche nach Tag-Schlüsseln. Geben Sie den Namen eines Schlüssels ein, um ihn in der Liste zu finden.

Verwenden von Tags mit dem AWS CLI

Mit der Lambda-API können Sie Tags zu vorhandenen Lambda-Ressourcen hinzufügen und entfernen, einschließlich Codesignaturkonfigurationen. Sie können beim Erstellen einer Codesignaturkonfiguration auch Tags hinzufügen, um eine Ressource während ihres gesamten Lebenszyklus mit Tags zu versehen.

Aktualisierung von Tags mit den Lambda-Tag-APIs

Sie können Tags für unterstützte Lambda-Ressourcen über die TagResourceund UntagResourceAPI-Operationen hinzufügen und entfernen.

Sie können diese Operationen mit der Taste AWS CLI aufrufen. Um Tags einer vorhandenen Ressource hinzuzufügen, verwenden Sie den folgenden `tag-resource`-Befehl. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Mit dem Befehl `untag-resource` können Sie Tags entfernen. In diesem Beispiel wird das Tag mit dem Schlüssel entfernt*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Hinzufügen von Tags beim Erstellen einer Codesignatur-Konfiguration

Verwenden Sie den CreateCodeSigningConfigAPI-Vorgang, um eine neue Lambda-Codesignaturkonfiguration mit Tags zu erstellen. Geben Sie den Parameter `Tags` an. Sie können diesen Vorgang mit dem `create-code-signing-config` AWS CLI Befehl und der `--tags` Option aufrufen. Weitere Informationen über diesen CLI-Befehl finden Sie unter create-code-signing-config in der AWS CLI -Befehlsreferenz.

Vergewissern Sie sich vor der Verwendung des `Tags`-Parameters mit `CreateCodeSigningConfig`, dass Ihre Rolle neben den üblichen Berechtigungen, die für diesen Vorgang erforderlich sind, auch die Berechtigung hat, Ressourcen zu taggen. Weitere Informationen zu den Berechtigungen für das Tagging finden Sie unter the section called “Erforderliche Berechtigungen zum Arbeiten mit Tags”.

Anzeigen von Tags mit den Lambda-Tag-APIs

Um die Tags anzuzeigen, die auf eine bestimmte Lambda-Ressource angewendet werden, verwenden Sie die `ListTags`-API-Operation. Weitere Informationen finden Sie unter ListTags.

Sie können diesen Vorgang mit dem `list-tags` AWS CLI Befehl aufrufen, indem Sie einen ARN (Amazon Resource Name) angeben.

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtern von Ressourcen nach Tag

Sie können den AWS Resource Groups Tagging API GetResourcesAPI-Vorgang verwenden, um Ihre Ressourcen nach Tags zu filtern. Die `GetResources`-Operation empfängt bis zu 10 Filter, wobei jeder Filter einen Tag-Schlüssel und bis zu 10 Tag-Werte enthält. Sie stellen `GetResources` mit einem `ResourceType` zur Verfügung, um nach bestimmten Ressourcentypen zu filtern.

Sie können diesen Vorgang mit dem `get-resources` AWS CLI Befehl aufrufen. Beispiele für die Verwendung von `get-resources` finden Sie unter get-resources in der AWS -CLI-Befehlsreferenz.

Verwenden von Lambda mit einem AWS SDK

AWS Software Development Kits (SDKs) sind für viele gängige Programmiersprachen verfügbar. Jedes SDK bietet eine API, Codebeispiele und Dokumentation, die es Entwicklern erleichtern, Anwendungen in ihrer bevorzugten Sprache zu erstellen.

SDK-Dokumentation	Codebeispiele
AWS SDK für C++	AWS SDK für C++ Code-Beispiele
AWS CLI	AWS CLI Code-Beispiele
AWS SDK für Go	AWS SDK für Go Code-Beispiele
AWS SDK für Java	AWS SDK für Java Code-Beispiele
AWS SDK für JavaScript	AWS SDK für JavaScript Code-Beispiele
AWS SDK für Kotlin	AWS SDK für Kotlin Code-Beispiele
AWS SDK für .NET	AWS SDK für .NET Code-Beispiele
AWS SDK für PHP	AWS SDK für PHP Code-Beispiele
AWS -Tools für PowerShell	AWS -Tools für PowerShell Code-Beispiele
AWS SDK für Python v4	AWS SDK für Python v4-Codebeispiele
AWS SDK für Python (Boto3)	AWS SDK für Python (Boto3) Codebeispiele
AWS SDK für Ruby	AWS SDK für Ruby Code-Beispiele
AWS SDK für Rust	AWS SDK für Rust Code-Beispiele
AWS SDK für SAP ABAP	AWS SDK für SAP ABAP Code-Beispiele
AWS SDK für Swift	AWS SDK für Swift Code-Beispiele

Für Beispiele, die sich speziell auf Lambda beziehen, siehe Codebeispiele für die Verwendung von Lambda AWS SDKs.

Beispiel für die Verfügbarkeit

Sie können nicht finden, was Sie brauchen? Fordern Sie ein Codebeispiel an, indem Sie unten den Link Provide feedback (Feedback geben) auswählen.

Codebeispiele für die Verwendung von Lambda AWS SDKs

Die folgenden Codebeispiele zeigen, wie Lambda mit einem verwendet wird AWS Softwareentwicklungskit (SDK).

Bei Grundlagen handelt es sich um Codebeispiele, die Ihnen zeigen, wie Sie die wesentlichen Vorgänge innerhalb eines Services ausführen.

Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Während Aktionen Ihnen zeigen, wie Sie einzelne Service-Funktionen aufrufen, können Sie Aktionen im Kontext der zugehörigen Szenarien anzeigen.

Szenarien sind Codebeispiele, die Ihnen zeigen, wie Sie bestimmte Aufgaben ausführen, indem Sie mehrere Funktionen innerhalb eines Service aufrufen oder mit anderen AWS-Services kombinieren.

AWS Beiträge der Gemeinschaft sind Beispiele, die von mehreren Teams auf der ganzen Welt erstellt wurden und verwaltet werden AWS. Um Feedback zu geben, verwenden Sie den Mechanismus, der in den verlinkten Repositorys bereitgestellt wird.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Codebeispiele mit SDK

- Grundlegende Beispiele für die Verwendung von Lambda AWS SDKs
  - Hallo Lambda
  - Lernen Sie die Grundlagen von Lambda mit einem AWS SDK
  - Aktionen für die Verwendung von Lambda AWS SDKs
    - Verwendung von CreateAlias mit einer CLI
    - Benutzen CreateFunction mit einem AWS SDK oder CLI
    - Verwendung von DeleteAlias mit einer CLI
    - Benutzen DeleteFunction mit einem AWS SDK oder CLI
    - Verwendung von DeleteFunctionConcurrency mit einer CLI
    - Verwendung von DeleteProvisionedConcurrencyConfig mit einer CLI
    - Verwendung von GetAccountSettings mit einer CLI
    - Verwendung von GetAlias mit einer CLI

    - Benutzen GetFunction mit einem AWS SDK oder CLI
    - Verwendung von GetFunctionConcurrency mit einer CLI
    - Verwendung von GetFunctionConfiguration mit einer CLI
    - Verwendung von GetPolicy mit einer CLI
    - Verwendung von GetProvisionedConcurrencyConfig mit einer CLI
    - Benutzen Aufrufen mit einem AWS SDK oder CLI
    - Benutzen ListFunctions mit einem AWS SDK oder CLI
    - Verwendung von ListProvisionedConcurrencyConfigs mit einer CLI
    - Verwendung von ListTags mit einer CLI
    - Verwendung von ListVersionsByFunction mit einer CLI
    - Verwendung von PublishVersion mit einer CLI
    - Verwendung von PutFunctionConcurrency mit einer CLI
    - Verwendung von PutProvisionedConcurrencyConfig mit einer CLI
    - Verwendung von RemovePermission mit einer CLI
    - Verwendung von TagResource mit einer CLI
    - Verwendung von UntagResource mit einer CLI
    - Verwendung von UpdateAlias mit einer CLI
    - Benutzen UpdateFunctionCode mit einem AWS SDK oder CLI
    - Benutzen UpdateFunctionConfiguration mit einem AWS SDK oder CLI
- Szenarien für die Verwendung von Lambda AWS SDKs
    - Bestätigen Sie automatisch bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion mithilfe einer AWS SDK
    - Migrieren Sie automatisch bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion mithilfe einer AWS SDK
    - Erstellen Sie eine API Gateway-REST-API zur Nachverfolgung COVID-19 Daten
    - Leihbibliothek-REST-API erstellen
    - Erstellen einer Messenger-Anwendung mit Step Functions
    - Erstellen Sie eine Rest-API mit Funktions-Proxy-Integration
    - Eine Anwendung für Foto-Asset-Management erstellen, mit der Benutzer Fotos mithilfe von Labels verwalten können
    - Erstellen einer Websocket-Chat-Anwendung mit API Gateway

- Erstellen einer Anwendung, die Kundenfeedback analysiert und Audio generiert
- Erstellen eines Monitoring-Dashboards mit einem Funktionsnamen als Variable
- Erstellen Sie Ihre erste serverlose Funktion
- Aufrufen einer Lambda-Funktion von einem Browser aus
- Transformieren von Daten für Ihre Anwendung mit S3 Object Lambda
- Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion
- Verwenden von Step Functions, um Lambda-Funktionen aufzurufen
- Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion
- Verwenden der Amazon Neptune-API zur Entwicklung einer Lambda-Funktion, die Graphdaten abfragt
- Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen
- Schreiben Sie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito-Benutzerauthentifizierung mithilfe einer AWS SDK

- Serverless-Beispiele für Lambda
  - Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion
  - Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser
  - Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser
  - Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser
  - Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser
  - Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser
  - Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen
  - Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser
  - Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser
  - Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser
  - Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser
- AWS Beiträge der Community für Lambda
  - Erstellen und Testen einer Serverless-Anwendung

Grundlegende Beispiele für die Verwendung von Lambda AWS SDKs

Die folgenden Codebeispiele zeigen, wie Sie die Grundlagen von verwenden AWS Lambda mit AWS SDKs.

Beispiele

- Hallo Lambda
- Lernen Sie die Grundlagen von Lambda mit einem AWS SDK
- Aktionen für die Verwendung von Lambda AWS SDKs
  - Verwendung von CreateAlias mit einer CLI
  - Benutzen CreateFunction mit einem AWS SDK oder CLI
  - Verwendung von DeleteAlias mit einer CLI
  - Benutzen DeleteFunction mit einem AWS SDK oder CLI
  - Verwendung von DeleteFunctionConcurrency mit einer CLI
  - Verwendung von DeleteProvisionedConcurrencyConfig mit einer CLI
  - Verwendung von GetAccountSettings mit einer CLI
  - Verwendung von GetAlias mit einer CLI
  - Benutzen GetFunction mit einem AWS SDK oder CLI
  - Verwendung von GetFunctionConcurrency mit einer CLI
  - Verwendung von GetFunctionConfiguration mit einer CLI
  - Verwendung von GetPolicy mit einer CLI
  - Verwendung von GetProvisionedConcurrencyConfig mit einer CLI
  - Benutzen Aufrufen mit einem AWS SDK oder CLI
  - Benutzen ListFunctions mit einem AWS SDK oder CLI
  - Verwendung von ListProvisionedConcurrencyConfigs mit einer CLI
  - Verwendung von ListTags mit einer CLI
  - Verwendung von ListVersionsByFunction mit einer CLI
  - Verwendung von PublishVersion mit einer CLI
  - Verwendung von PutFunctionConcurrency mit einer CLI
  - Verwendung von PutProvisionedConcurrencyConfig mit einer CLI

- Verwendung von RemovePermission mit einer CLI
- Verwendung von TagResource mit einer CLI
- Verwendung von UntagResource mit einer CLI
- Verwendung von UpdateAlias mit einer CLI
- Benutzen UpdateFunctionCode mit einem AWS SDK oder CLI
- Benutzen UpdateFunctionConfiguration mit einem AWS SDK oder CLI

Hallo Lambda

Die folgenden Codebeispiele zeigen, wie Sie mit der Verwendung von Lambda beginnen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {
```

```
 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Code für den CMakeLists.txt CMake-Datei.

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
```

```
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                 # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

Code für die Quelldatei „hello_lambda.cpp“.

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */
```

```
int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
 lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
 &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
 &functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
 std::endl;
                    std::cout << "   "
                              <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
```

```
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- API-Details finden Sie unter ListFunctions in AWS SDK für C++ API-Referenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)
```

```
// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(ctx, &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für JavaScript API-Referenz.

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import boto3


def main():
    """
    List the Lambda functions in your AWS account.
    """
    # Create the Lambda client
    lambda_client = boto3.client("lambda")

    # Use the paginator to list the functions
    paginator = lambda_client.get_paginator("list_functions")
    response_iterator = paginator.paginate()

    print("Here are the Lambda functions in your account:")
    for page in response_iterator:
        for function in page["Functions"]:
            print(f"  {function['FunctionName']}")


if __name__ == "__main__":
    main()
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
require 'aws-sdk-lambda'
```

```
# Creates an AWS Lambda client using the default credentials and configuration
def lambda_client
  Aws::Lambda::Client.new
end

# Lists the Lambda functions in your AWS account, paginating the results if
 necessary
def list_lambda_functions
  lambda = lambda_client

  # Use a pagination iterator to list all functions
  functions = []
  lambda.list_functions.each_page do |page|
    functions.concat(page.functions)
  end

  # Print the name and ARN of each function
  functions.each do |function|
    puts "Function name: #{function.function_name}"
    puts "Function ARN: #{function.function_arn}"
    puts
  end

  puts "Total functions: #{functions.count}"
end

list_lambda_functions if __FILE__ == $PROGRAM_NAME
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für Ruby API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Lernen Sie die Grundlagen von Lambda mit einem AWS SDK

Die folgenden Code-Beispiele veranschaulichen Folgendes:

- Erstellen Sie eine IAM-Rolle und eine Lambda-Funktion und laden Sie den Handlercode hoch.

- Rufen Sie die Funktion mit einem einzigen Parameter auf und erhalten Sie Ergebnisse.
- Aktualisieren Sie den Funktionscode und konfigurieren Sie mit einer Umgebungsvariablen.
- Rufen Sie die Funktion mit neuen Parametern auf und erhalten Sie Ergebnisse. Zeigt das zurückgegebene Ausführungsprotokoll an.
- Listen Sie die Funktionen für Ihr Konto auf und bereinigen Sie dann die Ressourcen.

Weitere Informationen zur Verwendung von Lambda finden Sie unter Erstellen einer Lambda-Funktion mit der Konsole.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Erstellen Sie Methoden, die Lambda-Aktionen ausführen.

```
namespace LambdaActions;

using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
```

```
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
```

```
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }


    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
 functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }


    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
```

```
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
       Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
 actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
 variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
```

```
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
 environmentVariables },
        };

        var response = await
 _lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

Erstellen Sie eine Funktion, die das Szenario ausführt.

```
global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;


using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
```

```
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
        var lambdaRoleWrapper =
 host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
```

```
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");

        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");

        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
            functionName,
            bucketName,
            incrementKey,
```

```
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
        DisplayFunctionList(functions);

        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));

        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
```

```
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        uiWrapper.DisplayTitle("Call updated function");
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;

            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
```

```
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
                {
                    Console.Write("Enter the first value: ");
                    value = Console.ReadLine();
                }
                while (value == string.Empty);

                string? value2;
                do
                {
                    Console.Write("Enter a second value: ");
                    value2 = Console.ReadLine();
                }
                while (value2 == string.Empty);

                functionParameters = "{" +
                    "\"action\": \"" + operation + "\", " +
                    "\"x\": \"" + value + "\"," +
                    "\"y\": \"" + value2 + "\"" +
                "}";
```

```
                answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
                Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
            }

            uiWrapper.PressEnter();
        } while (!done);

        // Delete the function created earlier.

        uiWrapper.DisplayTitle("Clean up resources");
        // Detach the IAM policy from the IAM role.
        Console.WriteLine("First detach the IAM policy from the role.");
        success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
        uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

        Console.WriteLine("Delete the AWS Lambda function.");
        success = await lambdaWrapper.DeleteFunctionAsync(functionName);
        if (success)
        {
            Console.WriteLine($"The {functionName} function was deleted.");
        }
        else
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
        }
        else
        {
            Console.WriteLine("Couldn't delete the role.");
        }
```

```
        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
```

```
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }

    /// <summary>
    /// Deletes an IAM role.
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
```

```
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;

public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
 permissions.");
        Console.WriteLine("\t3. Creates a Lambda function that increments the
 value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
 value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
 calculator.");
        Console.WriteLine("\t6. Calls the calculator function with the values
 entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
    }

    /// <summary>
    /// Display a message and wait until the user presses enter.
    /// </summary>
```

```
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
    {
        Console.WriteLine(SepBar);
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);

        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
```

```
        }

        PressEnter();
    }
}
```

Definieren Sie einen Lambda-Handler, der eine Zahl inkrementiert.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
        {
            return 0;
```

```
        }
    }
}
```

Definieren Sie einen zweiten Lambda-Handler, der arithmetische Operationen ausführt.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
        {
            case "add":
                result = x + y;
                break;
            case "subtract":
```

```
                result = x - y;
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für .NET - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
```

```
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
```

```
                    << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                                 invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
```

```
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
 Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
 std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
 return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }
```

```
        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
              << "This function uses an environment variable to control the logging
level."
              << std::endl;
    std::cout
              << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
DEBUG."
              << std::endl;
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
 an y."
            << std::endl;
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }
```

```
   // 6.  Invoke the updated Lambda function.
   do {
       int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
4 ", 1,
                                                  4);
       int x = askQuestionForInt("Enter an integer for the x value ");
       int y = askQuestionForInt("Enter an integer for the y value ");

       Aws::Utils::Json::JsonValue calculateJsonPayload;
       calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
       calculateJsonPayload.WithInteger("x", x);
       calculateJsonPayload.WithInteger("y", y);
       Aws::Lambda::Model::InvokeResult calculatedResult;
       if (invokeLambdaFunction(calculateJsonPayload,
                                Aws::Lambda::Model::LogType::Tail,
                                calculatedResult, client)) {
           Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
           Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                   jsonValue.View().GetAllObjects();
           auto iter = values.find("result");
           if (iter != values.end() && iter->second.IsIntegerType()) {
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsInteger() <<
std::endl;
           }
           else if (iter != values.end() && iter->second.IsFloatingPointType())
{
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsDouble() << std::endl;
           }
           else {
               std::cout << "There was an error in execution. Here is the log."
                         << std::endl;
               Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
               std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
           }
       }
```

```
        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
                          << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                          <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                  functionConfiguration.GetRuntime()) << ": "
                          << functionConfiguration.GetHandler()
                          << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
```

```
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                    << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);

    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
```

```
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }
```

```
            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für C++ -API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Erstellen Sie ein interaktives Szenario, das Ihnen zeigt, wie Sie mit Lambda-Funktionen loslegen können.

```
import (
 "archive/zip"
 "bytes"
 "context"
 "encoding/base64"
 "encoding/json"
 "errors"
 "fmt"
 "log"
 "os"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/iam"
 iamtypes "github.com/aws/aws-sdk-go-v2/service/iam/types"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/lambda/actions"
)

// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
```

```
//  3. Update the function code and configure with an environment variable.
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))

 role := scenario.GetOrCreateRole(ctx)
 funcName := scenario.CreateFunction(ctx, role)
 scenario.InvokeIncrement(ctx, funcName)
```

```
 scenario.UpdateFunction(ctx, funcName)
 scenario.InvokeCalculator(ctx, funcName)
 scenario.ListFunctions(ctx)
 scenario.Cleanup(ctx, role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole(ctx context.Context)
 *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(ctx, &iam.GetRoleInput{
  RoleName: aws.String(roleName)})
 if err != nil {
  var noSuch *iamtypes.NoSuchEntityException
  if errors.As(err, &noSuch) {
   log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
  } else {
   log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
    roleName, err)
  }
 } else {
  role = getOutput.Role
  log.Printf("Found role %v.\n", *role.RoleName)
 }
 if role == nil {
  trustPolicy := PolicyDocument{
   Version: "2012-10-17",
   Statement: []PolicyStatement{{
    Effect:    "Allow",
    Principal: map[string]string{"Service": "lambda.amazonaws.com"},
    Action:    []string{"sts:AssumeRole"},
```

```
   }},
  }
  policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
  createOutput, err := iamClient.CreateRole(ctx, &iam.CreateRoleInput{
   AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
   RoleName:                 aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
  }
  role = createOutput.Role
  _, err = iamClient.AttachRolePolicy(ctx, &iam.AttachRolePolicyInput{
   PolicyArn: aws.String(policyArn),
   RoleName:  aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
 err)
  }
  log.Printf("Created role %v.\n", *role.RoleName)
  log.Println("Let's give AWS a few seconds to propagate resources...")
  scenario.helper.Pause(10)
 }
 log.Println(strings.Repeat("-", 88))
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(ctx context.Context,
 role *iamtypes.Role) string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
 funcState := scenario.functionWrapper.CreateFunction(ctx, funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
```

```
 log.Println(strings.Repeat("-", 88))
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(ctx context.Context,
 funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, parameters,
 false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(ctx context.Context,
 funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
```

```
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(ctx, funcName,
 zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(ctx, funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(ctx context.Context,
 funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
  invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, calcParameters,
 true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
```

```
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions(ctx context.Context) {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(ctx, count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(ctx context.Context, role
 *iamtypes.Role, funcName string) {
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(ctx,
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
```

```
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(ctx, &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(ctx, &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(ctx, funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}

// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}
```

```
// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
 return buffer
}
```

Erstellen einer Struktur, die die einzelnen Lambda-Aktionen umschließt.

```
import (
```

```
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
```

```
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
```

```
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

```
// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
```

```
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}

// LambdaResultInt is used to deserialize an int result from a Lambda handler.
type LambdaResultInt struct {
 Result int `json:"result"`
}
```

```
// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

Definieren Sie einen Lambda-Handler, der eine Zahl inkrementiert.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

Definieren Sie einen zweiten Lambda-Handler, der arithmetische Operationen ausführt.

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
```

```
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/*
 *  Lambda function names appear as:
 *
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
including your credentials.
```

```
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = """

            Usage:
                <functionName> <role> <handler> <bucketName> <key>\s

            Where:
                functionName - The name of the Lambda function.\s
                role - The AWS Identity and Access Management (IAM) service role
 that has Lambda permissions.\s
                handler - The fully qualified method name (for example,
 example.Handler::handleRequest).\s
                bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
 name that contains the .zip or .jar used to update the Lambda function's code.\s
                key - The Amazon S3 key name that represents the .zip or .jar
 (for example, LambdaHello-1.0-SNAPSHOT.jar).
                """;

        if (args.length != 5) {
              System.out.println(usage);
              return;
        }
```

```
        String functionName = args[0];
        String role = args[1];
        String handler = args[2];
        String bucketName = args[3];
        String key = args[4];
        LambdaClient awsLambda = LambdaClient.builder()
            .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the AWS Lambda Basics scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("1. Create an AWS Lambda function.");
        String funArn = createLambdaFunction(awsLambda, functionName, key,
bucketName, role, handler);
        System.out.println("The AWS Lambda ARN is " + funArn);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
        getFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Update the Lambda function code and invoke it
again.");
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
```

```
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }

    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();
```

```
            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
```

```
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Invokes a specific AWS Lambda function.
     *
     * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to be invoked
     */
    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
        InvokeResponse res;
        try {
```

```
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            InvokeRequest request = InvokeRequest.builder()
                .functionName(functionName)
                .payload(payload)
                .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the code for an AWS Lambda function.
     *
     * @param awsLambda  the AWS Lambda client
     * @param functionName the name of the Lambda function to update
     * @param bucketName the name of the S3 bucket where the function code is
located
     * @param key the key (file name) of the function code in the S3 bucket
     * @throws LambdaException if there is an error updating the function code
     */
    public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
        try {
            LambdaWaiter waiter = awsLambda.waiter();
            UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();
```

```
            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Deletes an AWS Lambda function.
     *
     * @param awsLambda     an instance of the {@link LambdaClient} class, which
 is used to interact with the AWS Lambda service
     * @param functionName  the name of the Lambda function to be deleted
     *
     * @throws LambdaException if an error occurs while deleting the Lambda
 function
     */
    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                .functionName(functionName)
                .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK for Java 2.x - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Erstellen Sie eine AWS Identity and Access Management (IAM) -Rolle, die Lambda die Berechtigung erteilt, in Protokolle zu schreiben.

```
    logger.log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

Erstellen Sie eine Lambda-Funktion und laden Sie Handlercode hoch.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
```

```
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

Rufen Sie die Funktion mit einem einzigen Parameter auf und erhalten Sie Ergebnisse.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

Aktualisieren Sie den Funktionscode und konfigurieren Sie seine Lambda-Umgebung mit einer Umgebungsvariablen.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
```

```
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};

const updateFunctionConfiguration = async (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = await client.send(command);
  await waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

Listen Sie die Funktionen für Ihr Konto auf.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

Löschen Sie die IAM-Rolle für Ihre Lambda-Funktion.

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
```

```
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für JavaScript -API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
            <functionName> <role> <handler> <bucketName> <updatedBucketName>
 <key>
```

```
       Where:
           functionName - The name of the AWS Lambda function.
           role - The AWS Identity and Access Management (IAM) service role that
has AWS Lambda permissions.
           handler - The fully qualified method name (for example,
example.Handler::handleRequest).
           bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the ZIP or JAR used for the Lambda function's code.
           updatedBucketName - The Amazon S3 bucket name that contains the .zip
or .jar used to update the Lambda function's code.
           key - The Amazon S3 key name that represents the .zip or .jar file
(for example, LambdaHello-1.0-SNAPSHOT.jar).
           """

   if (args.size != 6) {
       println(usage)
       exitProcess(1)
   }

   val functionName = args[0]
   val role = args[1]
   val handler = args[2]
   val bucketName = args[3]
   val updatedBucketName = args[4]
   val key = args[5]

   println("Creating a Lambda function named $functionName.")
   val funArn = createScFunction(functionName, bucketName, key, handler, role)
   println("The AWS Lambda ARN is $funArn")

   // Get a specific Lambda function.
   println("Getting the $functionName AWS Lambda function.")
   getFunction(functionName)

   // List the Lambda functions.
   println("Listing all AWS Lambda functions.")
   listFunctionsSc()

   // Invoke the Lambda function.
   println("*** Invoke the Lambda function.")
   invokeFunctionSc(functionName)

   // Update the AWS Lambda function code.
```

```
    println("*** Update the Lambda function code.")
    updateFunctionCode(functionName, updatedBucketName, key)

    // println("*** Invoke the function again after updating the code.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function configuration.
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
```

```
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
    val functionRequest =
        GetFunctionRequest {
            functionName = functionNameVal
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
```

```
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
    functionNameVal: String?,
    bucketName: String?,
    key: String?,
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?,
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {
```

```
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS -SDK für Kotlin.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;
```

```
class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("--------------------------------------\n");
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("--------------------------------------\n");

        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";
```

```
        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "amzn-s3-demo-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
```

```
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
        $invokeFunction = $lambdaService->invoke($functionName,
 $calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
```

```
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
"Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
$listLambdaFunctions['Functions']);
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
```

```
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für PHP - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Definieren Sie einen Lambda-Handler, der eine Zahl inkrementiert.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

Definieren Sie einen zweiten Lambda-Handler, der arithmetische Operationen ausführt.

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}
```

```
def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

Erstellen Sie Funktionen, die Lambda-Aktionen umschließen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource
```

```
    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
buffer. This
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
        :param destination_file: The name to give the file when it's deployed to
Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.

        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
```

```
                raise
        return role

    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
a
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
        :return: The role and a value that indicates whether the role is newly
created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version":"2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
iam_role_name)
            else:
```

```
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
                    iam_role_name,
                    policy_arn,
                )
                raise

        return role, True

    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
```

```
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

```
    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
```

```
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
```

```
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

Erstellen Sie eine Funktion, die das Szenario ausführt.

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda
 handler.
```

```
    :param calculator_file: The name of the file that contains the calculator
Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such
as the
                        IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the AWS Lambda getting started with functions demo.")
    print("-" * 88)

    wrapper = LambdaWrapper(lambda_client, iam_resource)

    print("Checking for IAM role for Lambda...")
    iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
    if should_wait:
        logger.info("Giving AWS time to create resources...")
        wait(10)

    print(f"Looking for function {lambda_name}...")
    function = wrapper.get_function(lambda_name)
    if function is None:
        print("Zipping the Python script into a deployment package...")
        deployment_package = wrapper.create_deployment_package(
            basic_file, f"{lambda_name}.py"
        )
        print(f"...and creating the {lambda_name} Lambda function.")
        wrapper.create_function(
            lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
        )
    else:
        print(f"Function {lambda_name} already exists.")
    print("-" * 88)

    print(f"Let's invoke {lambda_name}. This function increments a number.")
    action_params = {
        "action": "increment",
        "number": q.ask("Give me a number to increment: ", q.is_int),
    }
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
```

```
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
            f"Calculating {action_params['x']} {action_params['action']}
{action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
```

```
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
 q.is_yesno)
    print("-" * 88)

    if q.ask(
        "Do you want to list all of the functions in your account? (y/n) ",
 q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS SDK für Python (Boto3).
  - CreateFunction

- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Richten Sie die erforderlichen IAM-Berechtigungen für eine Lambda-Funktion ein, die Protokolle schreiben kann.

```
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
# @param action: Whether to create or destroy the IAM apparatus.
# @return: The IAM role.
def manage_iam(iam_role_name, action)
  case action
  when 'create'
    create_iam_role(iam_role_name)
  when 'destroy'
    destroy_iam_role(iam_role_name)
  else
    raise "Incorrect action provided. Must provide 'create' or 'destroy'"
  end
end

private

def create_iam_role(iam_role_name)
```

```
    role_policy = {
      'Version': '2012-10-17',
      'Statement': [
        {
          'Effect': 'Allow',
          'Principal': { 'Service': 'lambda.amazonaws.com' },
          'Action': 'sts:AssumeRole'
        }
      ]
    }
    role = @iam_client.create_role(
      role_name: iam_role_name,
      assume_role_policy_document: role_policy.to_json
    )
    @iam_client.attach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    wait_for_role_to_exist(iam_role_name)
    @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
    sleep(10)
    [role, role_policy.to_json]
  end

  def destroy_iam_role(iam_role_name)
    @iam_client.detach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    @iam_client.delete_role(role_name: iam_role_name)
    @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
  end

  def wait_for_role_to_exist(iam_role_name)
    @iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
```

```
  end
```

Definieren Sie einen Lambda-Handler, der eine als Aufrufparameter bereitgestellte Zahl inkrementiert.

```
require 'logger'

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  log_level = ENV['LOG_LEVEL']
  logger.level = case log_level
                 when 'debug'
                   Logger::DEBUG
                 when 'info'
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug('This is a debug log message.')
  logger.info('This is an info log message. Code executed successfully!')
  number = event['number'].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

Komprimieren (ZIP) Sie Ihre Lambda-Funktion in ein Bereitstellungspaket.

```
  # Creates a Lambda deployment package in .zip format.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
```

```
 def create_deployment_package(source_file)
   Dir.chdir(File.dirname(__FILE__))
   if File.exist?('lambda_function.zip')
     File.delete('lambda_function.zip')
     @logger.debug('Deleting old zip: lambda_function.zip')
   end
   Zip::File.open('lambda_function.zip', create: true) do |zipfile|
     zipfile.add('lambda_function.rb', "#{source_file}.rb")
   end
   @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
   File.read('lambda_function.zip').to_s
 rescue StandardError => e
   @logger.error("There was an error creating deployment package:\n
#{e.message}")
 end
```

Erstellen Sie eine neue Lambda-Funktion.

```
 # Deploys a Lambda function.
 #
 # @param function_name: The name of the Lambda function.
 # @param handler_name: The fully qualified name of the handler function.
 # @param role_arn: The IAM role to use for the function.
 # @param deployment_package: The deployment package that contains the function
code in .zip format.
 # @return: The Amazon Resource Name (ARN) of the newly created function.
 def create_function(function_name, handler_name, role_arn, deployment_package)
   response = @lambda_client.create_function({
                                               role: role_arn.to_s,
                                               function_name: function_name,
                                               handler: handler_name,
                                               runtime: 'ruby2.7',
                                               code: {
                                                 zip_file: deployment_package
                                               },
                                               environment: {
                                                 variables: {
                                                   'LOG_LEVEL' => 'info'
                                                 }
                                               }
                                             })
```

```
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

Rufen Sie Ihre Lambda-Funktion mit optionalen Laufzeitparametern auf.

```
  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

Aktualisieren Sie die Konfiguration Ihrer Lambda-Funktion, um eine neue Umgebungsvariable einzufügen.

```
  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
```

```
                                                    }
                                                  }
                                                })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

Aktualisieren Sie den Code Ihrer Lambda-Funktion mit einem anderen Bereitstellungspaket, das anderen Code enthält.

```
  # Updates the code for a Lambda function by submitting a .zip archive that
contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
```

```
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

Listen Sie alle vorhandenen Lambda-Funktionen mithilfe des eingebauten Paginators auf.

```
# Lists the Lambda functions for the current account.
def list_functions
  functions = []
  @lambda_client.list_functions.each do |response|
    response['functions'].each do |function|
      functions.append(function['function_name'])
    end
  end
  functions
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error listing functions:\n #{e.message}")
end
```

Löschen Sie eine bestimmte Lambda-Funktion.

```
# Deletes a Lambda function.
# @param function_name: The name of the function to delete.
def delete_function(function_name)
  print "Deleting function: #{function_name}..."
  @lambda_client.delete_function(
    function_name: function_name
  )
  print 'Done!'.green
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
end
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Ruby - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction

- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Das Cargo.toml mit Abhängigkeiten, die in diesem Szenario verwendet werden.

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
```

```
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

Eine Sammlung von Hilfsprogrammen, die das Aufrufen von Lambda für dieses Szenario optimieren. Diese Datei ist src/ations.rs in der Kiste.

```
use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{fmt::Display, path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
```

```
}

impl FromStr for Operation {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl Display for Operation {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Operation::Plus => write!(f, "plus"),
            Operation::Minus => write!(f, "minus"),
            Operation::Times => write!(f, "times"),
            Operation::DividedBy => write!(f, "divided-by"),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
```

```
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
                map.serialize_key(&"i".to_string())?;
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}

// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
```

```
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);

impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
 name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
 bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
 Option<String>) -> Self {
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
 "rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
        let (bucket, own_bucket) =
            match bucket {
```

```
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
                    })),
                    true,
                ),
            };

        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()

.location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
            OwnBucket(own_bucket),
        )
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
```

```
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }

    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
 anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
```

```
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
            .create_role()
```

```
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
            .send()
            .await;

        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
        match self.get_function().await {
```

```
            Ok(func) => {
                if let Some(config) = func.configuration() {
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
                    }
```

```
                }
            }
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
        }
        Ok(false)
    }

    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
```

```
            .map_err(anyhow::Error::from)
    }

    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
```

```
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }

    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
```

```
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }

    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };
```

```
        (delete_function, delete_bucket)
    }
}

/**
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

Eine Binärdatei, um das Szenario vom Anfang bis Ende auszuführen, wobei Befehlszeilen-Flags verwendet werden, um einige Verhaltensweisen zu steuern. Diese Datei ist src/bin/scenario.rs in der Kiste.

```
/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
```

```
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.


The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
```

```
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
```

```
    LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,

    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}
```

```
fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
                    "trace".to_string(),
                )])))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
```

```
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
```

```
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS -SDK für Rust.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   TRY.
       "Create an AWS Identity and Access Management (IAM) role that grants AWS
Lambda permission to write to logs."
       DATA(lv_policy_document) = `{` &&
           `"Version":"2012-10-17",       ` &&
```

```
                  `"Statement": [` &&
                    `{` &&
                      `"Effect": "Allow",` &&
                      `"Action": [` &&
                        `"sts:AssumeRole"` &&
                      `],` &&
                      `"Principal": {` &&
                        `"Service": [` &&
                          `"lambda.amazonaws.com"` &&
                        `]` &&
                      `}` &&
                    `}` &&
                  `]` &&
                `}`.
        TRY.
            DATA(lo_create_role_output) = lo_iam->createrole(
                    iv_rolename = iv_role_name
                    iv_assumerolepolicydocument = lv_policy_document
                    iv_description = 'Grant lambda permission to write to
 logs' ).
            DATA(lv_role_arn) = lo_create_role_output->get_role( )->get_arn( ).
            MESSAGE 'IAM role created.' TYPE 'I'.
            WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
 ready for use. "
          CATCH /aws1/cx_iamentityalrdyexex.
            DATA(lo_role) = lo_iam->getrole( iv_rolename = iv_role_name ).
            lv_role_arn = lo_role->get_role( )->get_arn( ).
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
```

```
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lv_role_arn
                 iv_handler = iv_handler
                 io_code = io_initial_zip_file
                 iv_description = 'AWS Lambda code example' ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "increment",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_initial_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
```

```
                        iv_payload = lv_json ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
```

```
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables ) ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_updated_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
```

```
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'W'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.
```

```
        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS -SDK für SAP ABAP.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Definieren Sie die erste Lambda-Funktion, die einfach den angegebenen Wert erhöht.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "increment",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "increment",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)

import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
```

```
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}


/// The Lambda function body.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let incrementLambdaRuntime = LambdaRuntime {
        (event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?

    if action != "increment" {
        context.logger.error("Unrecognized operation: \"\(action)\". The only
 supported action is \"increment\".")
    } else {
        answer = event.number + 1
        context.logger.info("The calculated answer is \(answer!).")
    }

    let response = Response(answer: answer)
    return response
}

// Run the Lambda runtime code.
```

```
try await incrementLambdaRuntime.run()
```

Definieren Sie die zweite Lambda-Funktion, die eine arithmetische Operation mit zwei Zahlen ausführt.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "calculator",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "calculator",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)
```

```
import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

/// A dictionary mapping operation names to closures that perform that
/// operation and return the result.
let actions = [
    "plus": { (x: Int, y: Int) -> Int in
        return x + y
    },
    "minus": { (x: Int, y: Int) -> Int in
        return x - y
    },
    "times": { (x: Int, y: Int) -> Int in
        return x * y
    },
    "divided-by": { (x: Int, y: Int) -> Int in
        return x / y
    }
]

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}



/// The Lambda function's entry point. Called by the Lambda runtime.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
```

```
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let calculatorLambdaRuntime = LambdaRuntime {
        (_ event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?
    var actionFunc: ((Int, Int) -> Int)?

    // Get the closure to run to perform the calculation.

    actionFunc = await actions[action]

    guard let actionFunc else {
        context.logger.error("Unrecognized operation '\(action)\'")
        return Response(answer: nil)
    }

    // Perform the calculation and return the answer.

    answer = actionFunc(event.x, event.y)

    guard let answer else {
        context.logger.error("Error computing \(event.x) \(action) \(event.y)")
    }
    context.logger.info("\(event.x) \(action) \(event.y) = \(answer)")

    return Response(answer: answer)
}

try await calculatorLambdaRuntime.run()
```

Definieren Sie das Hauptprogramm, das die beiden Lambda-Funktionen aufruft.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
```

```
// build this package.

import PackageDescription

let package = Package(
    name: "lambda-basics",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/awslabs/aws-sdk-swift",
            from: "1.0.0"),
        .package(
            url: "https://github.com/apple/swift-argument-parser.git",
            branch: "main"
        )
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "lambda-basics",
            dependencies: [
                .product(name: "AWSLambda", package: "aws-sdk-swift"),
                .product(name: "AWSIAM", package: "aws-sdk-swift"),
                .product(name: "ArgumentParser", package: "swift-argument-
parser")
            ],
            path: "Sources"
        )
    ]
)

//
/// An example demonstrating a variety of important AWS Lambda functions.

import ArgumentParser
import AWSIAM
import SmithyWaitersAPI
```

```
import AWSClientRuntime
import AWSLambda
import Foundation

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct IncrementRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

struct Response: Encodable, Decodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}

struct CalculatorRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

let exampleName = "SwiftLambdaRoleExample"
let basicsFunctionName = "lambda-basics-function"

/// The ARN of the standard IAM policy for execution of Lambda functions.
let policyARN = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

struct ExampleCommand: ParsableCommand {
    // -MARK: Command arguments
    @Option(help: "Name of the IAM Role to use for the Lambda functions")
    var role = exampleName
    @Option(help: "Zip archive containing the 'increment' lambda function")
    var incpath: String
    @Option(help: "Zip archive containing the 'calculator' lambda function")
    var calcpath: String
    @Option(help: "Name of the Amazon S3 Region to use (default: us-east-1)")
```

```
    var region = "us-east-1"

    static var configuration = CommandConfiguration(
        commandName: "lambda-basics",
        abstract: """
        This example demonstrates several common operations using AWS Lambda.
        """,
        discussion: """
        """
    )

    /// Returns the specified IAM role object.
    ///
    /// - Parameters:
    ///   - iamClient: `IAMClient` to use when looking for the role.
    ///   - roleName: The name of the role to check.
    ///
    /// - Returns: The `IAMClientTypes.Role` representing the specified role.
    func getRole(iamClient: IAMClient, roleName: String) async throws
                -> IAMClientTypes.Role {
        do {
            let roleOutput = try await iamClient.getRole(
                input: GetRoleInput(
                    roleName: roleName
                )
            )

            guard let role = roleOutput.role else {
                throw ExampleError.roleNotFound
            }
            return role
        } catch {
            throw ExampleError.roleNotFound
        }
    }

    /// Create the AWS IAM role that will be used to access AWS Lambda.
    ///
    /// - Parameters:
    ///   - iamClient: The AWS `IAMClient` to use.
    ///   - roleName: The name of the AWS IAM role to use for Lambda.
    ///
    /// - Throws: `ExampleError.roleCreateError`
    ///
```

```
    /// - Returns: The `IAMClientTypes.Role` struct that describes the new role.
    func createRoleForLambda(iamClient: IAMClient, roleName: String) async throws
-> IAMClientTypes.Role {
        let output = try await iamClient.createRole(
            input: CreateRoleInput(
                assumeRolePolicyDocument:
                """
                {
                    "Version":"2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {"Service": "lambda.amazonaws.com"},
                            "Action": "sts:AssumeRole"
                        }
                    ]
                }
                """,
                roleName: roleName
            )
        )

        guard let role = output.role else {
            throw ExampleError.roleCreateError
        }

        // Wait for the role to be ready for use.

        _ = try await iamClient.waitUntilRoleExists(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetRoleInput(roleName: roleName)
        )

        return role
    }

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
```

```
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
-> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }

    /// Create the specified AWS Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to create.
    ///   - roleArn: The ARN of the role to apply to the function.
    ///   - path: The path of the Zip archive containing the function.
    ///
    /// - Returns: `true` if the AWS Lambda was successfully created; `false`
    ///   if it wasn't.
    func createFunction(lambdaClient: LambdaClient, functionName: String,
                                roleArn: String?, path: String) async throws ->
Bool {
        do {
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
```

```
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }

        // Wait for a while to be sure the function is done being created.

        let output = try await lambdaClient.waitUntilFunctionActiveV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(functionName: functionName)
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Update the AWS Lambda function with new code to run when the function
    /// is invoked.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to update.
    ///   - path: The pathname of the Zip file containing the packaged Lambda
    ///     function.
    /// - Throws: `ExampleError.zipFileReadError`
    /// - Returns: `true` if the function's code is updated successfully.
    ///   Otherwise, returns `false`.
    func updateFunctionCode(lambdaClient: LambdaClient, functionName: String,
                            path: String) async throws -> Bool {
        let zipUrl = URL(fileURLWithPath: path)
```

```
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
        }

        let output = try await lambdaClient.waitUntilFunctionUpdatedV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(
                functionName: functionName
            )
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Tell the server-side component to log debug output by setting its
```

```
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
async throws {
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
```

```
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
```

```
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
Int {
        do {
            let incRequest = IncrementRequest(action: "increment", number:
number)
            let incData = try! JSONEncoder().encode(incRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: incData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }

    /// Invoke the calculator Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - action: Which arithmetic operation to perform: "plus", "minus",
    ///     "times", or "divided-by".
    ///   - x: The first number to use in the computation.
    ///   - y: The second number to use in the computation.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: The computed answer as an `Int`.
    func invokeCalculator(lambdaClient: LambdaClient, action: String, x: Int, y:
Int) async throws -> Int {
        do {
```

```
            let calcRequest = CalculatorRequest(action: action, x: x, y: y)
            let calcData = try! JSONEncoder().encode(calcRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: calcData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }

    }

    /// Perform the example's tasks.
    func basics() async throws {
        let iamClient = try await IAMClient(
            config: IAMClient.IAMClientConfig(region: region)
        )

        let lambdaClient = try await LambdaClient(
            config: LambdaClient.LambdaClientConfig(region: region)
        )

        /// The IAM role to use for the example.
        var iamRole: IAMClientTypes.Role

        // Look for the specified role. If it already exists, use it. If not,
        // create it and attach the desired policy to it.

        do {
            iamRole = try await getRole(iamClient: iamClient, roleName: role)
```

```
        } catch ExampleError.roleNotFound {
            // The role wasn't found, so create it and attach the needed
            // policy.

            iamRole = try await createRoleForLambda(iamClient: iamClient,
roleName: role)

            do {
                _ = try await iamClient.attachRolePolicy(
                    input: AttachRolePolicyInput(policyArn: policyARN, roleName:
role)
                )
            } catch {
                throw ExampleError.policyError
            }
        }

        // Give the policy time to attach to the role.

        sleep(5)

        // Look to see if the function already exists. If it does, throw an
        // error.

        if await doesLambdaFunctionExist(lambdaClient: lambdaClient, name:
basicsFunctionName) {
            throw ExampleError.functionAlreadyExists
        }

        // Create, then invoke, the "increment" version of the calculator
        // function.

        print("Creating the increment Lambda function...")
        if try await createFunction(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                    roleArn: iamRole.arn, path: incpath) {
            print("Running increment function calls...")
            for number in 0...4 {
                do {
                    let answer = try await invokeIncrement(lambdaClient:
lambdaClient, number: number)
                    print("Increment \(number) = \(answer)")
                } catch {
```

```
                    print("Error incrementing \(number): ",
error.localizedDescription)
                }
            }
        } else {
            print("*** Failed to create the increment function.")
        }

        // Enable debug logging.

        print("\nEnabling debug logging...")
        try await enableDebugLogging(lambdaClient: lambdaClient, functionName:
basicsFunctionName)

        // Change it to a basic arithmetic calculator. Then invoke it a few
        // times.

        print("\nReplacing the Lambda function with a calculator...")

        if try await updateFunctionCode(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                        path: calcpath) {
            print("Running calculator function calls...")
            for x in [6, 10] {
                for y in [2, 4] {
                    for action in ["plus", "minus", "times", "divided-by"] {
                        do {
                            let answer = try await invokeCalculator(lambdaClient:
lambdaClient, action: action, x: x, y: y)
                            print("\(x) \(action) \(y) = \(answer)")
                        } catch {
                            print("Error calculating \(x) \(action) \(y): ",
error.localizedDescription)
                        }
                    }
                }
            }
        }

        // List all lambda functions.

        let functionNames = try await getFunctionNames(lambdaClient:
lambdaClient)
```

```
        if functionNames.count > 0 {
            print("\nAWS Lambda functions available on your account:")
            for name in functionNames {
                print("  \(name)")
            }
        }

        // Delete the lambda function.

        print("Deleting lambda function...")

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }

        // Detach the role from the policy, then delete the role.

        print("Deleting the AWS IAM role...")

        do {
            _ = try await iamClient.detachRolePolicy(
                input: DetachRolePolicyInput(
                    policyArn: policyARN,
                    roleName: role
                )
            )
            _ = try await iamClient.deleteRole(
                input: DeleteRoleInput(
                    roleName: role
                )
            )
        } catch {
            throw ExampleError.deleteRoleError
        }
    }
}

// -MARK: - Entry point
```

```
/// The program's asynchronous entry point.
@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.basics()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}


/// Errors thrown by the example's functions.
enum ExampleError: Error {
    /// An AWS Lambda function with the specified name already exists.
    case functionAlreadyExists
    /// The specified role doesn't exist.
    case roleNotFound
    /// Unable to create the role.
    case roleCreateError
    /// Unable to delete the role.
    case deleteRoleError
    /// Unable to attach a policy to the role.
    case policyError
    /// Unable to get the executable directory.
    case executableNotFound
    /// An error occurred creating a lambda function.
    case createLambdaError
    /// An error occurred invoking the lambda function.
    case invokeError
    /// No answer received from the invocation.
    case noAnswerReceived
    /// Unable to list the AWS Lambda functions.
    case listFunctionsError
    /// Unable to update the AWS Lambda function.
    case updateFunctionError
    /// Unable to update the function configuration.
    case updateFunctionConfigurationError
    /// The environment response is missing after an
```

```
    /// UpdateEnvironmentConfiguration attempt.
    case environmentResponseMissingError
    /// The environment variables are missing from the EnvironmentResponse and
    /// no errors occurred.
    case environmentVariablesMissingError
    /// The log level is incorrect after attempting to set it.
    case logLevelIncorrectError
    /// Unable to load the AWS Lambda function's Zip file.
    case zipFileReadError

    var errorDescription: String? {
        switch self {
        case .functionAlreadyExists:
            return "An AWS Lambda function with that name already exists."
        case .roleNotFound:
            return "The specified role doesn't exist."
        case .deleteRoleError:
            return "Unable to delete the AWS IAM role."
        case .roleCreateError:
            return "Unable to create the specified role."
        case .policyError:
            return "An error occurred attaching the policy to the role."
        case .executableNotFound:
            return "Unable to find the executable program directory."
        case .createLambdaError:
            return "An error occurred creating a lambda function."
        case .invokeError:
            return "An error occurred invoking a lambda function."
        case .noAnswerReceived:
            return "No answer received from the lambda function."
        case .listFunctionsError:
            return "Unable to list the AWS Lambda functions."
        case .updateFunctionError:
            return "Unable to update the AWS lambda function."
        case .updateFunctionConfigurationError:
            return "Unable to update the AWS lambda function configuration."
        case .environmentResponseMissingError:
            return "The environment is missing from the response after updating
the function configuration."
        case .environmentVariablesMissingError:
            return "While no error occurred, no environment variables were
returned following function configuration."
        case .logLevelIncorrectError:
```

```
            return "The log level is incorrect after attempting to set it to
 DEBUG."
        case .zipFileReadError:
            return "Unable to read the AWS Lambda function."
        }
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS SDK für Swift.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aktionen für die Verwendung von Lambda AWS SDKs

Die folgenden Codebeispiele zeigen, wie Sie einzelne Lambda-Aktionen ausführen mit AWS SDKs. Jedes Beispiel enthält einen Link zu GitHub, wo Sie Anweisungen zum Einrichten und Ausführen des Codes finden.

Diese Auszüge rufen die Lambda-API auf und sind Codeauszüge aus größeren Programmen, die im Kontext ausgeführt werden müssen. Sie können Aktionen im Kontext unter Szenarien für die Verwendung von Lambda AWS SDKs anzeigen.

Die folgenden Beispiele enthalten nur die am häufigsten verwendeten Aktionen. Eine vollständige Liste finden Sie in der AWS Lambda -API-Referenz.

Beispiele

- Verwendung von CreateAlias mit einer CLI

- Benutzen CreateFunction mit einem AWS SDK oder CLI
- Verwendung von DeleteAlias mit einer CLI
- Benutzen DeleteFunction mit einem AWS SDK oder CLI
- Verwendung von DeleteFunctionConcurrency mit einer CLI
- Verwendung von DeleteProvisionedConcurrencyConfig mit einer CLI
- Verwendung von GetAccountSettings mit einer CLI
- Verwendung von GetAlias mit einer CLI
- Benutzen GetFunction mit einem AWS SDK oder CLI
- Verwendung von GetFunctionConcurrency mit einer CLI
- Verwendung von GetFunctionConfiguration mit einer CLI
- Verwendung von GetPolicy mit einer CLI
- Verwendung von GetProvisionedConcurrencyConfig mit einer CLI
- Benutzen Aufrufen mit einem AWS SDK oder CLI
- Benutzen ListFunctions mit einem AWS SDK oder CLI
- Verwendung von ListProvisionedConcurrencyConfigs mit einer CLI
- Verwendung von ListTags mit einer CLI
- Verwendung von ListVersionsByFunction mit einer CLI
- Verwendung von PublishVersion mit einer CLI
- Verwendung von PutFunctionConcurrency mit einer CLI
- Verwendung von PutProvisionedConcurrencyConfig mit einer CLI
- Verwendung von RemovePermission mit einer CLI
- Verwendung von TagResource mit einer CLI
- Verwendung von UntagResource mit einer CLI
- Verwendung von UpdateAlias mit einer CLI
- Benutzen UpdateFunctionCode mit einem AWS SDK oder CLI
- Benutzen UpdateFunctionConfiguration mit einem AWS SDK oder CLI

Verwendung von **`CreateAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `CreateAlias` verwendet wird.

CLI

AWS CLI

So erstellen Sie einen Alias für eine Lambda-Funktion

Im folgenden Beispiel für `create-alias` wird ein Alias namens `LIVE` erstellt, der auf Version 1 der `my-function`-Lambda-Funktion verweist.

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

Ausgabe:

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "Description": "alias for live version of function"
}
```

Weitere Informationen finden Sie unter Konfiguration AWS Aliase für Lambda-Funktionen in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter CreateAlias in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird ein neuer Lambda-Alias für die angegebene Version und Routing-Konfiguration erstellt, um den Prozentsatz der Aufrufe für dieses System anzugeben.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- API-Details finden Sie unter CreateAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird ein neuer Lambda-Alias für die angegebene Version und Routing-Konfiguration erstellt, um den Prozentsatz der Aufrufe für dieses System anzugeben.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- API-Details finden Sie unter CreateAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`CreateFunction`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `CreateFunction` verwendet wird.

Aktionsbeispiele sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Sie können diese Aktion in den folgenden Codebeispielen im Kontext sehen:

- Kennenlernen der Grundlagen
- Erstellen Sie eine Rest-API mit Funktions-Proxy-Integration
- Erstellen eines Überwachungs-Dashboards mit einem Funktionsnamen als Variable
- Erstellen Sie Ihre erste serverlose Funktion
- Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/// <summary>
/// Creates a new Lambda function.
/// </summary>
/// <param name="functionName">The name of the function.</param>
/// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
/// bucket where the zip file containing the code is located.</param>
/// <param name="s3Key">The Amazon S3 key of the zip file.</param>
/// <param name="role">The Amazon Resource Name (ARN) of a role with the
/// appropriate Lambda permissions.</param>
/// <param name="handler">The name of the handler function.</param>
/// <returns>The Amazon Resource Name (ARN) of the newly created
/// Lambda function.</returns>
public async Task<string> CreateLambdaFunctionAsync(
    string functionName,
    string s3Bucket,
    string s3Key,
    string role,
    string handler)
{
    // Defines the location for the function code.
    // S3Bucket - The S3 bucket where the file containing
    //            the source code is stored.
    // S3Key    - The name of the file containing the code.
    var functionCode = new FunctionCode
    {
        S3Bucket = s3Bucket,
        S3Key = s3Key,
    };

    var createFunctionRequest = new CreateFunctionRequest
    {
        FunctionName = functionName,
```

```
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);
```

```
        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);
```

```
        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für C++ API-Referenz.

CLI

AWS CLI

So erstellen Sie eine Lambda-Funktion

Im folgenden Beispiel für `create-function` wird eine Lambda-Funktion mit dem Namen `my-function` erstellt.

```
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs22.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

Inhalt von `my-function.zip`:

```
This file is a deployment package that contains your function code and any
 dependencies.
```

Ausgabe:

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2025-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs22.x",
    "Description": ""
}
```

Weitere Informationen finden Sie unter Konfigurieren des Speichers der Lambda-Funktion im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter CreateFunction in AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
```

```
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
```

```
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
```

```
                                                String functionName,
                                                String key,
                                                String bucketName,
                                                String role,
                                                String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für JavaScript API-Referenz.

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für die Kotlin-API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
                'Handler' => "$handler.lambda_handler",
            ]);
        });
    }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für PHP API-Referenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel erstellt eine neue C#-Funktion (dotnetcore1.0 Runtime) mit dem MyFunction Namenin AWS Lambda, das die kompilierten Binärdateien für die Funktion aus

einer Zip-Datei im lokalen Dateisystem bereitstellt (relative oder absolute Pfade können verwendet werden). C#-Lambda-Funktionen spezifizieren den Handler für die Funktion anhand der Bezeichnung AssemblyName::Namespace.ClassName::MethodName. Sie sollten die Teile Assembly-Name (ohne .dll-Suffix), Namespace, Klassenname und Methodenname der Handler-Spezifikation entsprechend ersetzen. Für die neue Funktion werden die Umgebungsvariablen 'envvar1' und 'envvar2' aus den bereitgestellten Werten eingerichtet.

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

Ausgabe:

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, mit dem Unterschied, dass die Funktions-Binärdateien zunächst in einen Amazon S3-Bucket hochgeladen werden (der sich in derselben Region befinden muss wie die vorgesehene Lambda-Funktion) und das resultierende S3-Objekt dann bei der Erstellung der Funktion referenziert wird.

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
```

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel erstellt eine neue C#-Funktion (dotnetcore1.0 Runtime) mit dem MyFunction Namenin AWS Lambda, das die kompilierten Binärdateien für die Funktion aus einer Zip-Datei im lokalen Dateisystem bereitstellt (relative oder absolute Pfade können verwendet werden). C#-Lambda-Funktionen spezifizieren den Handler für die Funktion anhand der Bezeichnung AssemblyName::Namespace.ClassName::MethodName. Sie sollten die Teile Assembly-Name (ohne .dll-Suffix), Namespace, Klassenname und Methodenname der Handler-Spezifikation entsprechend ersetzen. Für die neue Funktion werden die Umgebungsvariablen 'envvar1' und 'envvar2' aus den bereitgestellten Werten eingerichtet.

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

Ausgabe:

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
```

```
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, mit dem Unterschied, dass die Funktions-Binärdateien zunächst in einen Amazon S3-Bucket hochgeladen werden (der sich in derselben Region befinden muss wie die vorgesehene Lambda-Funktion) und das resultierende S3-Objekt dann bei der Erstellung der Funktion referenziert wird.

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
```

```
        self.iam_resource = iam_resource


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
 code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
```

```
                                                zip_file: deployment_package
                                              },
                                              environment: {
                                                variables: {
                                                  'LOG_LEVEL' => 'info'
                                                }
                                              }
                                            })
   @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
   response
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error creating #{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * Create a function, uploading from a zip file.
    */
   pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
       let code = self.prepare_function(zip_file, None).await?;

       let key = code.s3_key().unwrap().to_string();
```

```
        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));
```

```
        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für Rust-API-Referenz.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    TRY.
        lo_lmd->createfunction(
            iv_functionname = iv_function_name
            iv_runtime = `python3.9`
            iv_role = iv_role_arn
            iv_handler = iv_handler
            io_code = io_zip_file
            iv_description = 'AWS Lambda code example' ).
        MESSAGE 'Lambda function created.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
```

```
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für SAP ABAP API-Referenz.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            // Read the Zip archive containing the AWS Lambda function.
```

```
            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }
```

- Einzelheiten zur API finden Sie unter CreateFunction in AWS SDK für Swift API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`DeleteAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteAlias` verwendet wird.

CLI

AWS CLI

So löschen Sie einen Alias einer Lambda-Funktion

Im folgenden Beispiel für `delete-alias` wird der Alias mit dem Namen `LIVE` aus der `my-function`-Lambda-Funktion gelöscht.

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Konfiguration AWS Aliase für Lambda-Funktionen in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter DeleteAlias in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird die im Befehl erwähnte Lambda-Funktion Alias gelöscht.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- API-Details finden Sie unter DeleteAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird die im Befehl erwähnte Lambda-Funktion Alias gelöscht.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- API-Details finden Sie unter DeleteAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`DeleteFunction`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteFunction` verwendet wird.

Aktionsbeispiele sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Sie können diese Aktion in den folgenden Codebeispielen im Kontext sehen:

- Kennenlernen der Grundlagen
- Erstellen Sie eine Rest-API mit Funktions-Proxy-Integration
- Erstellen eines Überwachungs-Dashboards mit einem Funktionsnamen als Variable
- Erstellen Sie Ihre erste serverlose Funktion
- Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);

    if (outcome.IsSuccess()) {
        std::cout << "The lambda function was successfully deleted." <<
std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Beispiel 1: So löschen Sie eine Lambda-Funktion anhand des Funktionsnamens

Im folgenden Beispiel für `delete-function` wird die Lambda-Funktion `my-function` durch Angabe des Funktionsnamens gelöscht.

```
aws lambda delete-function \
    --function-name my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Beispiel 2: Eine Lambda-Funktion anhand des Funktions-ARN löschen

Im folgenden Beispiel für `delete-function` wird die Lambda-Funktion `my-function` durch Angabe des ARN der Funktion gelöscht.

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Beispiel 3: Eine Lambda-Funktion anhand eines teilweisen Funktions-ARN löschen

Im folgenden Beispiel für `delete-function` wird die Lambda-Funktion `my-function` durch Angabe des teilweisen ARN der Funktion gelöscht.

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * Deletes an AWS Lambda function.
    *
    * @param awsLambda     an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName  the name of the Lambda function to be deleted
    *
    * @throws LambdaException if an error occurs while deleting the Lambda
function
    */
   public static void deleteLambdaFunction(LambdaClient awsLambda, String
functionName) {
       try {
           DeleteFunctionRequest request = DeleteFunctionRequest.builder()
               .functionName(functionName)
               .build();

           awsLambda.deleteFunction(request);
           System.out.println("The " + functionName + " function was deleted");

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für JavaScript API-Referenz.

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
```

```
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für die Kotlin-API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für PHP API-Referenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird eine bestimmte Version einer Lambda-Funktion gelöscht

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- API-Details finden Sie unter DeleteFunction in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird eine bestimmte Version einer Lambda-Funktion gelöscht

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- API-Details finden Sie unter DeleteFunction in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print 'Done!'.green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
   pub async fn delete_function(
       &self,
       location: Option<String>,
   ) -> (
       Result<DeleteFunctionOutput, anyhow::Error>,
       Result<DeleteRoleOutput, anyhow::Error>,
       Option<Result<DeleteObjectOutput, anyhow::Error>>,
   ) {
       info!("Deleting lambda function {}", self.lambda_name);
       let delete_function = self
           .lambda_client
           .delete_function()
           .function_name(self.lambda_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       info!("Deleting iam role {}", self.role_name);
       let delete_role = self
           .iam_client
           .delete_role()
           .role_name(self.role_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
           if let Some(location) = location {
               info!("Deleting object {location}");
               Some(
```

```
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für Rust-API-Referenz.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    TRY.
        lo_lmd->deletefunction( iv_functionname = iv_function_name ).
        MESSAGE 'Lambda function deleted.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für SAP ABAP API-Referenz.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }
```

- Einzelheiten zur API finden Sie unter DeleteFunction in AWS SDK für Swift API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`DeleteFunctionConcurrency`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteFunctionConcurrency` verwendet wird.

CLI

AWS CLI

So entfernen Sie das Limit für die reservierte gleichzeitige Ausführung einer Funktion

Im folgenden `delete-function-concurrency`-Beispiel wird das reservierte Limit für gleichzeitige Ausführung aus der `my-function`-Funktion gelöscht.

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Gleichzeitigkeit für eine Lambda-Funktion reservieren im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter DeleteFunctionConcurrency in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Funktionsgleichzeitigkeit der Lambda-Funktion aufgehoben.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- API-Details finden Sie unter DeleteFunctionConcurrency in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird die Funktionsgleichzeitigkeit der Lambda-Funktion aufgehoben.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- API-Details finden Sie unter DeleteFunctionConcurrency in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`DeleteProvisionedConcurrencyConfig`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteProvisionedConcurrencyConfig` verwendet wird.

CLI

AWS CLI

So löschen Sie eine bereitgestellte Gleichzeitigkeitskonfiguration

Das folgende `delete-provisioned-concurrency-config`-Beispiel löscht die bereitgestellte Gleichzeitigkeitskonfiguration für den `GREEN`-Alias der angegebenen Funktion.

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität für eine Event-Funktion in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter DeleteProvisionedConcurrencyConfig in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für einen bestimmten Alias entfernt.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- API-Details finden Sie unter DeleteProvisionedConcurrencyConfig in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für einen bestimmten Alias entfernt.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- API-Details finden Sie unter DeleteProvisionedConcurrencyConfig in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetAccountSettings`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetAccountSettings` verwendet wird.

CLI

AWS CLI

Um Details zu Ihrem Konto in einem abzurufen AWS Region

Im folgenden `get-account-settings`-Beispiel werden die Lambda-Grenzwerte und Nutzungsinformationen für Ihr Konto angezeigt.

```
aws lambda get-account-settings
```

Ausgabe:

```
{
```

```
    "AccountLimit": {
        "CodeSizeUnzipped": 262144000,
        "UnreservedConcurrentExecutions": 1000,
        "ConcurrentExecutions": 1000,
        "CodeSizeZipped": 52428800,
        "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
        "FunctionCount": 4,
        "TotalCodeSize": 9426
    }
}
```

Weitere Informationen finden Sie unter AWS Lambda Limits im AWS -Lambda-Entwicklerhandbuch.

- API-Details finden Sie unter GetAccountSettings in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel wird angezeigt, um das Kontolimit und die Kontonutzung zu vergleichen

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Ausgabe:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- API-Details finden Sie unter GetAccountSettings in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel wird angezeigt, um das Kontolimit und die Kontonutzung zu vergleichen

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Ausgabe:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- API-Details finden Sie unter GetAccountSettings in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetAlias` verwendet wird.

CLI

AWS CLI

So rufen Sie Details über einen Funktionsalias ab

Das folgende `get-alias`-Beispiel zeigt Details für den Alias namens `LIVE` für die `my-function`-Lambda-Funktion an.

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

Ausgabe:

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
```

```
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "Description": "alias for live version of function"
}
```

Weitere Informationen finden Sie unter Konfiguration AWS Aliase für Lambda-Funktionen in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter GetAlias in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel werden die Gewichtungen der Routing-Konfiguration für einen bestimmten Lambda-Funktionsalias abgerufen.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Ausgabe:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- API-Details finden Sie unter GetAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel werden die Gewichtungen der Routing-Konfiguration für einen bestimmten Lambda-Funktionsalias abgerufen.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Ausgabe:

```
AdditionalVersionWeights
------------------------
```

```
{[1, 0.6]}
```

- API-Details finden Sie unter GetAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`GetFunction`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `GetFunction` verwendet wird.

Aktionsbeispiele sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Sie können diese Aktion in den folgenden Codebeispielen im Kontext sehen:

- Kennenlernen der Grundlagen
- Erstellen Sie Ihre erste serverlose Funktion

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
```

```
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::GetFunctionRequest request;
       request.SetFunctionName(functionName);

       Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

       if (outcome.IsSuccess()) {
           std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                     << std::endl;
       }
       else {
           std::cerr << "Error with Lambda::GetFunction. "
```

```
                        << outcome.GetError().GetMessage()
                        << std::endl;
        }
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für C++ API-Referenz.

CLI

AWS CLI

So rufen Sie Informationen über eine Funktion ab

Das folgende Beispiel für `get-function` zeigt Informationen zur Funktion `my-function` an.

```
aws lambda get-function \
    --function-name  my-function
```

Ausgabe:

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
```

```
        "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
        "Timeout": 3,
        "LastModified": "2025-09-24T18:20:35.054+0000",
        "Runtime": "nodejs22.x",
        "Description": ""
    }
}
```

Weitere Informationen finden Sie unter Konfigurieren des Speichers der Lambda-Funktion im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter GetFunction in AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)
```

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * Retrieves information about an AWS Lambda function.
     *
```

```
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
 is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
 information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
 awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
 response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für PHP API-Referenz.

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
```

```
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client
```

```
  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
```

```
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für Rust-API-Referenz.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    TRY.
        oo_result = lo_lmd->getfunction( iv_functionname = iv_function_name ).
    " oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für SAP ABAP API-Referenz.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
 -> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }
```

- Einzelheiten zur API finden Sie unter GetFunction in AWS SDK für Swift API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetFunctionConcurrency`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetFunctionConcurrency` verwendet wird.

CLI

AWS CLI

So zeigen Sie die reservierte Gleichzeitigkeitseinstellung für eine Funktion an

Das folgende `get-function-concurrency`-Beispiel ruft die reservierte Gleichzeitigkeitseinstellung für die angegebene Funktion ab.

```
aws lambda get-function-concurrency \
    --function-name my-function
```

Ausgabe:

```
{
    "ReservedConcurrentExecutions": 250
}
```

Weitere Informationen finden Sie unter Konfiguration der reservierten Parallelität für eine Event-Funktion in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter GetFunctionConcurrency in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel ruft die reservierte Gleichzeitigkeit für die Lambda-Funktion ab

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Ausgabe:

```
ReservedConcurrentExecutions
----------------------------
100
```

- API-Details finden Sie unter GetFunctionConcurrency in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel ruft die reservierte Gleichzeitigkeit für die Lambda-Funktion ab

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Ausgabe:

```
ReservedConcurrentExecutions
----------------------------
100
```

- API-Details finden Sie unter GetFunctionConcurrency in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetFunctionConfiguration`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetFunctionConfiguration` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Erstellen Sie Ihre erste serverlose Funktion

CLI

AWS CLI

So rufen Sie die versionsspezifischen Einstellungen einer Lambda-Funktion ab

Das folgende `get-function-configuration`-Beispiel zeigt die Einstellungen für Version 2 der `my-function`-Funktion.

```
aws lambda get-function-configuration \
```

```
    --function-name  my-function:2
```

Ausgabe:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "MemorySize": 256,
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter GetFunctionConfiguration in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel gibt die versionsspezifische Konfiguration einer Lambda-Funktion zurück.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Ausgabe:

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                              :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- API-Details finden Sie unter GetFunctionConfiguration in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel gibt die versionsspezifische Konfiguration einer Lambda-Funktion zurück.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Ausgabe:

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                              :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- API-Details finden Sie unter GetFunctionConfiguration in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetPolicy`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetPolicy` verwendet wird.

CLI

AWS CLI

So rufen Sie die ressourcenbasierte IAM-Richtlinie für eine Funktion, eine Version oder einen Alias ab

Das folgende `get-policy`-Beispiel zeigt Richtlinieninformationen über die `my-function`-Lambda-Funktion an.

```
aws lambda get-policy \
    --function-name my-function
```

Ausgabe:

```
{
    "Policy": "{\"Version\":\"2012-10-17\",\"Id\":\"default\",\"Statement
\":[{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":{\"Service\":
\"iotevents.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-west-2:123456789012:function:my-function\"}]}",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

Weitere Informationen finden Sie unter Verwenden Resource-based Richtlinien für AWS Lambda in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter GetPolicy in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Funktionsrichtlinie der Lambda-Funktion angezeigt

```
Get-LMPolicy -FunctionName test -Select Policy
```

Ausgabe:

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- API-Details finden Sie unter GetPolicy in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird die Funktionsrichtlinie der Lambda-Funktion angezeigt

```
Get-LMPolicy -FunctionName test -Select Policy
```

Ausgabe:

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- API-Details finden Sie unter GetPolicy in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetProvisionedConcurrencyConfig`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetProvisionedConcurrencyConfig` verwendet wird.

CLI

AWS CLI

So zeigen Sie eine bereitgestellte Gleichzeitigkeitskonfiguration an

Im folgenden `get-provisioned-concurrency-config`-Beispiel werden Details für die bereitgestellte Gleichzeitigkeitskonfiguration für den BLUE-Alias der angegebenen Funktion angezeigt.

```
aws lambda get-provisioned-concurrency-config \
```

```
    --function-name my-function \
    --qualifier BLUE
```

Ausgabe:

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität für eine Event-Funktion in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter GetProvisionedConcurrencyConfig in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für den angegebenen Alias der Lambda-Funktion abgerufen.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Ausgabe:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- API-Details finden Sie unter GetProvisionedConcurrencyConfig in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für den angegebenen Alias der Lambda-Funktion abgerufen.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Ausgabe:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- API-Details finden Sie unter GetProvisionedConcurrencyConfig in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`Aufrufen`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `Invoke` verwendet wird.

Aktionsbeispiele sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Sie können diese Aktion in den folgenden Codebeispielen im Kontext sehen:

- Kennenlernen der Grundlagen
- Erstellen eines Überwachungs-Dashboards mit einem Funktionsnamen als Variable
- Erstellen Sie Ihre erste serverlose Funktion
- Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für .NET -API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für C++ -API-Referenz.

CLI

AWS CLI

Beispiel 1: Eine Lambda-Funktion synchron aufrufen

Im folgenden Beispiel für `invoke` wird die Funktion `my-function` synchron aufgerufen. Das `cli-binary-format` Die Option ist erforderlich, wenn Sie verwenden AWS CLI-Version 2. Weitere Informationen finden Sie unter Von der AWS CLI unterstützte globale Befehlszeilenoptionen im AWS -CLI-Benutzerhandbuch.

```
aws lambda invoke response.json \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }'
```

Ausgabe:

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

Weitere Informationen finden Sie unter Synchrones Aufrufen einer Lambda-Funktion im Entwicklerhandbuch für AWS Lambda.

Beispiel 2: Eine Lambda-Funktion asynchron aufrufen

Im folgenden Beispiel für `invoke` wird die Funktion `my-function` asynchron aufgerufen. Die `cli-binary-format` Die Option ist erforderlich, wenn Sie verwenden AWS CLI-Version 2. Weitere Informationen finden Sie unter Von der AWS CLI unterstützte globale Befehlszeilenoptionen im AWS -CLI-Benutzerhandbuch.

```
aws lambda invoke response.json \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
```

```
    --payload '{ "name": "Bob" }'
```

Ausgabe:

```
{
    "StatusCode": 202
}
```

Weitere Informationen finden Sie unter Asynchrones Aufrufen einer Lambda-Funktion im Entwicklerhandbuch für AWS Lambda.

- Weitere API-Informationen finden Sie unter Invoke in der AWS CLI -Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
```

```
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für Go -API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * Invokes a specific AWS Lambda function.
    *
    * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
    * @param functionName the name of the AWS Lambda function to be invoked
    */
   public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
       InvokeResponse res;
       try {
           // Need a SdkBytes instance for the payload.
           JSONObject jsonObj = new JSONObject();
           jsonObj.put("inputValue", "2000");
           String json = jsonObj.toString();
           SdkBytes payload = SdkBytes.fromUtf8String(json);

           InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

           res = awsLambda.invoke(request);
           String value = res.payload().asUtf8String();
           System.out.println(value);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK for Java 2.x -API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für JavaScript -API-Referenz.

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
```

```
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- Weitere API-Informationen finden Sie unter Invoke in der API-Referenz zum AWS -SDK für Kotlin.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für PHP -API-Referenz.

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
 dict
                                is serialized to JSON before it is sent to
 Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
 included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- Weitere API-Informationen finden Sie unter Invoke in der API-Referenz zum AWS -SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für Ruby -API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
 anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
```

- Weitere API-Informationen finden Sie unter Aufrufen in der API-Referenz zum AWS -SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   TRY.
       DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
         `{`  &&
           `"action": "increment",`  &&
           `"number": 10` &&
         `}` ).
       oo_result = lo_lmd->invoke(                 " oo_result is returned for
testing purposes. "
                  iv_functionname = iv_function_name
                  iv_payload = lv_json ).
       MESSAGE 'Lambda function invoked.' TYPE 'I'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvrequestcontex.
       MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidzipfileex.
       MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
     CATCH /aws1/cx_lmdrequesttoolargeex.
       MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
     CATCH /aws1/cx_lmdunsuppedmediatyp00.
```

```
        MESSAGE 'Invoke request body does not have JSON as its content type.'
TYPE 'E'.
   ENDTRY.
```

- Weitere API-Informationen finden Sie unter Invoke (Aufrufen) in der API-Referenz für das AWS -SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
 Int {
        do {
            let incRequest = IncrementRequest(action: "increment", number:
 number)
            let incData = try! JSONEncoder().encode(incRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
```

```
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: incData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }
```

- Detaillierte API-Informationen finden Sie unter Aufrufen in der API-Referenz zum AWS SDK für Swift.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`ListFunctions`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `ListFunctions` verwendet wird.

Aktionsbeispiele sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Sie können diese Aktion in den folgenden Codebeispielen im Kontext sehen:

- Kennenlernen der Grundlagen
- Erstellen eines Überwachungs-Dashboards mit einem Funktionsnamen als Variable

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/// <summary>
/// Get a list of Lambda functions.
/// </summary>
/// <returns>A list of FunctionConfiguration objects.</returns>
public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
{
    var functionList = new List<FunctionConfiguration>();

    var functionPaginator =
        _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
    await foreach (var function in functionPaginator.Functions)
    {
        functionList.Add(function);
    }

    return functionList;
}
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

   std::vector<Aws::String> functions;
   Aws::String marker;

   do {
       Aws::Lambda::Model::ListFunctionsRequest request;
       if (!marker.empty()) {
           request.SetMarker(marker);
       }

       Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
               request);

       if (outcome.IsSuccess()) {
           const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
           std::cout << result.GetFunctions().size()
                     << " lambda functions were retrieved." << std::endl;

           for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
               functions.push_back(functionConfiguration.GetFunctionName());
               std::cout << functions.size() << "  "
                         << functionConfiguration.GetDescription() << std::endl;
               std::cout << "   "
                         <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                 functionConfiguration.GetRuntime()) << ": "
                         << functionConfiguration.GetHandler()
                         << std::endl;
           }
           marker = result.GetNextMarker();
       }
       else {
           std::cerr << "Error with Lambda::ListFunctions. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
```

```
        }
    } while (!marker.empty());
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Eine Liste der Lambda-Funktionen abrufen

Im folgenden Beispiel für `list-functions` wird eine Liste aller Funktionen für den aktuellen Benutzer angezeigt.

```
aws lambda list-functions
```

Ausgabe:

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2025-09-23T18:32:33.857+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
```

```
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2025-10-01T16:47:28.490+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
                {
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
```

```
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
            "CodeSize": 299,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2025-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

Weitere Informationen finden Sie unter Konfigurieren des Speichers der Lambda-Funktion im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter ListFunctions in AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
```

```
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für Go API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
```

```
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für PHP API-Referenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel werden alle Lambda-Funktionen mit sortierter Codegröße angezeigt

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Ausgabe:

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- API-Details finden Sie unter ListFunctions in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel werden alle Lambda-Funktionen mit sortierter Codegröße angezeigt

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Ausgabe:

```
FunctionName                                                  Runtime   Timeout
 CodeSize
------------                                                  -------   -------
 --------
test                                                          python2.7       3
   243
MylambdaFunction123                                           python3.8     600
   659
myfuncpython1                                                 python3.8     303
   675
```

- API-Details finden Sie unter ListFunctions in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
```

```
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
```

```
    @lambda_client.list_functions.each do |response|
      response['functions'].each do |function|
        functions.append(function['function_name'])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error listing functions:\n #{e.message}")
  end
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für Rust-API-Referenz.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   TRY.
       oo_result = lo_lmd->listfunctions( ).       " oo_result is returned for
testing purposes. "
       DATA(lt_functions) = oo_result->get_functions( ).
       MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für SAP ABAP API-Referenz.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
```

```
import AWSLambda
import Foundation

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }
```

- Einzelheiten zur API finden Sie unter ListFunctions in AWS SDK für Swift API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`ListProvisionedConcurrencyConfigs`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `ListProvisionedConcurrencyConfigs` verwendet wird.

CLI

AWS CLI

So erhalten Sie eine Liste der bereitgestellten Gleichzeitigkeitskonfigurationen

Das folgende `list-provisioned-concurrency-configs`-Beispiel listet die bereitgestellten Gleichzeitigkeitskonfigurationen für die angegebene Funktion auf.

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

Ausgabe:

```
{
    "ProvisionedConcurrencyConfigs": [
        {
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:GREEN",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität für eine Event-Funktion in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter ListProvisionedConcurrencyConfigs in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Liste der bereitgestellten Gleichzeitigkeitskonfigurationen für eine Lambda-Funktion abgerufen.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- API-Details finden Sie unter ListProvisionedConcurrencyConfigs in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird die Liste der bereitgestellten Gleichzeitigkeitskonfigurationen für eine Lambda-Funktion abgerufen.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- API-Details finden Sie unter ListProvisionedConcurrencyConfigs in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`ListTags`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `ListTags` verwendet wird.

CLI

AWS CLI

So rufen Sie die Liste der Tags für eine Lambda-Funktion ab

Das folgende `list-tags`-Beispiel zeigt die an die `my-function`-Lambda-Funktion angehängten Tags an.

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Ausgabe:

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

Weitere Informationen finden Sie unter Tagging von Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter ListTags in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Ruft die Tags und ihre Werte ab, die derzeit für die angegebene Funktion festgelegt sind.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Ausgabe:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- API-Details finden Sie unter ListTags in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Ruft die Tags und ihre Werte ab, die derzeit für die angegebene Funktion festgelegt sind.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Ausgabe:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- API-Details finden Sie unter ListTags in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`ListVersionsByFunction`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `ListVersionsByFunction` verwendet wird.

CLI

AWS CLI

So rufen Sie eine Liste der Versionen einer Funktion ab

Im folgenden `list-versions-by-function`-Beispiel wird die Liste der Versionen für die `my-function`-Lambda-Funktion angezeigt.

```
aws lambda list-versions-by-function \
    --function-name my-function
```

Ausgabe:

```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
```

```
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
            "CodeSize": 304,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
```

```
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

Weitere Informationen finden Sie unter Konfiguration AWS Aliase für Lambda-Funktionen in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter ListVersionsByFunction in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel gibt die Liste der versionsspezifischen Konfigurationen für jede Version der Lambda-Funktion zurück.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Ausgabe:

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- API-Details finden Sie unter ListVersionsByFunction in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel gibt die Liste der versionsspezifischen Konfigurationen für jede Version der Lambda-Funktion zurück.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Ausgabe:

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- API-Details finden Sie unter ListVersionsByFunction in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`PublishVersion`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `PublishVersion` verwendet wird.

CLI

AWS CLI

So veröffentlichen Sie eine neue Version einer Funktion

Im folgenden Beispiel für `publish-version` wird eine neue Version der `my-function`-Lambda-Funktion veröffentlicht.

```
aws lambda publish-version \
    --function-name my-function
```

Ausgabe:

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
    "CodeSize": 294,
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

Weitere Informationen finden Sie unter Konfiguration AWS Aliase für Lambda-Funktionen in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter PublishVersion in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird eine Version für den vorhandenen Snapshot von Lambda-Funktionscode erstellt

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- API-Details finden Sie unter PublishVersion in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird eine Version für den vorhandenen Snapshot von Lambda-Funktionscode erstellt

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- API-Details finden Sie unter PublishVersion in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`PutFunctionConcurrency`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `PutFunctionConcurrency` verwendet wird.

CLI

AWS CLI

So konfigurieren Sie eine reservierte Gleichzeitigkeitsgrenze für eine Funktion

Im folgenden `put-function-concurrency`-Beispiel werden 100 reservierte gleichzeitige Ausführungen für die `my-function`-Funktion konfiguriert.

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

Ausgabe:

```
{
    "ReservedConcurrentExecutions": 100
}
```

Weitere Informationen finden Sie unter Gleichzeitigkeit für eine Lambda-Funktion reservieren im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter PutFunctionConcurrency in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel werden die Gleichzeitigkeitseinstellungen für die Funktion als Ganzes angewendet.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- API-Details finden Sie unter PutFunctionConcurrency in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel werden die Gleichzeitigkeitseinstellungen für die Funktion als Ganzes angewendet.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- API-Details finden Sie unter PutFunctionConcurrency in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **PutProvisionedConcurrencyConfig** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `PutProvisionedConcurrencyConfig` verwendet wird.

CLI

AWS CLI

So weisen Sie bereitgestellte Gleichzeitigkeit zu

Das folgende `put-provisioned-concurrency-config`-Beispiel weist dem BLUE-Alias der angegebenen Funktion 100 bereitgestellte Gleichzeitigkeit zu.

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

Ausgabe:

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität für eine Event-Funktion in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter PutProvisionedConcurrencyConfig in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: In diesem Beispiel wird dem Alias einer Funktion eine bereitgestellte Gleichzeitigkeitskonfiguration hinzugefügt

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- API-Details finden Sie unter PutProvisionedConcurrencyConfig in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: In diesem Beispiel wird dem Alias einer Funktion eine bereitgestellte Gleichzeitigkeitskonfiguration hinzugefügt

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- API-Details finden Sie unter PutProvisionedConcurrencyConfig in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`RemovePermission`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `RemovePermission` verwendet wird.

CLI

AWS CLI

So entfernen Sie Berechtigungen aus einer vorhandenen Lambda-Funktion

Das folgende `remove-permission`-Beispiel entfernt die Berechtigung zum Aufrufen einer Funktion namens `my-function`.

```
aws lambda remove-permission \
```

```
    --function-name my-function \
    --statement-id sns
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Verwenden Resource-based Richtlinien für AWS Lambda in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter RemovePermission in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel entfernt die Funktionsrichtlinie für die angegebene StatementId einer Lambda-Funktion.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- API-Details finden Sie unter RemovePermission in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel entfernt die Funktionsrichtlinie für die angegebene StatementId einer Lambda-Funktion.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- API-Details finden Sie unter RemovePermission in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`TagResource`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `TagResource` verwendet wird.

CLI

AWS CLI

So fügen Sie einer vorhandenen Lambda-Funktion Tags hinzu

Das folgende `tag-resource`-Beispiel fügt der angegebenen Lambda-Funktion ein Tag mit dem Schlüsselnamen `DEPARTMENT` und einem Wert von `Department A` hinzu.

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Tagging von Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter TagResource in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Fügt die drei Tags (Washington, Oregon und Kalifornien) und ihre zugehörigen Werte zu der angegebenen Funktion hinzu, die durch ihre ARN identifiziert wird.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- API-Details finden Sie unter TagResource in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Fügt die drei Tags (Washington, Oregon und Kalifornien) und ihre zugehörigen Werte zu der angegebenen Funktion hinzu, die durch ihre ARN identifiziert wird.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- API-Details finden Sie unter TagResource in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`UntagResource`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `UntagResource` verwendet wird.

CLI

AWS CLI

So entfernen Sie Tags aus einer vorhandenen Lambda-Funktion

Das folgende `untag-resource`-Beispiel entfernt das Tag mit dem Schlüsselnamen `DEPARTMENT`-Tag aus der Lambda-Funktion `my-function`.

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Tagging von Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter UntagResource in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Es wird ein einziger Aufruf an den Dienst gesendet, um die Tags zu entfernen.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Beispiel 2: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Sobald der Dienst pro bereitgestelltem Tag aufgerufen wurde.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- API-Details finden Sie unter UntagResource in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Es wird ein einziger Aufruf an den Dienst gesendet, um die Tags zu entfernen.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Beispiel 2: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Sobald der Dienst pro bereitgestelltem Tag aufgerufen wurde.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- API-Details finden Sie unter UntagResource in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`UpdateAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `UpdateAlias` verwendet wird.

CLI

AWS CLI

So aktualisieren Sie einen Funktionsalias

Das folgende `update-alias`-Beispiel aktualisiert den Alias mit dem Namen `LIVE` so, dass er auf Version 3 der `my-function`-Lambda-Funktion verweist.

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

Ausgabe:

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "Description": "alias for live version of function"
}
```

Weitere Informationen finden Sie unter Konfiguration AWS Aliase für Lambda-Funktionen in der AWS Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter UpdateAlias in AWS CLI Befehlsreferenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel aktualisiert die Konfiguration einer vorhandenen Lambda-Funktion-Alias. Es aktualisiert die RoutingConfiguration Wert, um 60% (0,6) des Datenverkehrs auf Version 1 zu verlagern

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- API-Details finden Sie unter UpdateAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel aktualisiert die Konfiguration einer vorhandenen Lambda-Funktion-Alias. Es aktualisiert die RoutingConfiguration Wert, um 60% (0,6) des Datenverkehrs auf Version 1 zu verlagern

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- API-Details finden Sie unter UpdateAlias in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`UpdateFunctionCode`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `UpdateFunctionCode` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);
```

```
        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Den Code einer Lambda-Funktion aktualisieren

Im folgenden Beispiel für `update-function-code` wird der Code der unveröffentlichten Version ($LATEST) der Funktion `my-function` durch den Inhalt der angegebenen ZIP-Datei ersetzt.

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

Ausgabe:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "MemorySize": 256,
    "Version": "$LATEST",
```

```
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * Updates the code for an AWS Lambda function.
    *
    * @param awsLambda  the AWS Lambda client
    * @param functionName the name of the Lambda function to update
    * @param bucketName the name of the S3 bucket where the function code is
located
    * @param key the key (file name) of the function code in the S3 bucket
    * @throws LambdaException if there is an error updating the function code
    */
   public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
       try {
           LambdaWaiter waiter = awsLambda.waiter();
           UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();

           UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
```

```
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
```

```
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
    {
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für PHP API-Referenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Aktualisiert die Funktion mit dem Namen "MyFunctionmit neuem Inhalt, der in der angegebenen Zip-Datei enthalten ist. Für eine C#.NET-Core-Lambda-Funktion sollte die ZIP-Datei die kompilierte Assembly enthalten.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, verwendet aber ein Amazon S3-Objekt, das den aktualisierten Code enthält, um die Funktion zu aktualisieren.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- API-Details finden Sie unter UpdateFunctionCode in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Aktualisiert die Funktion mit dem Namen "MyFunctionmit neuem Inhalt, der in der angegebenen Zip-Datei enthalten ist. Für eine C#.NET-Core-Lambda-Funktion sollte die ZIP-Datei die kompilierte Assembly enthalten.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, verwendet aber ein Amazon S3-Objekt, das den aktualisierten Code enthält, um die Funktion zu aktualisieren.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- API-Details finden Sie unter UpdateFunctionCode in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
```

```
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the code for a Lambda function by submitting a .zip archive that
 contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
 #{function_name}:\n #{e.message}")
```

```
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;
```

```
        Ok(update)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für Rust-API-Referenz.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   TRY.
       oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
             iv_functionname = iv_function_name
             iv_zipfile = io_zip_file ).

       MESSAGE 'Lambda function code updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodestorageexcdex.
       MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für SAP ABAP API-Referenz.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        let zipUrl = URL(fileURLWithPath: path)
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
        }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode in AWS SDK für Swift API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Benutzen **`UpdateFunctionConfiguration`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `UpdateFunctionConfiguration` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
```

```
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Die Konfiguration einer Funktion ändern

Im folgenden Beispiel für `update-function-configuration` wird die Speichergröße für die unveröffentlichte Version ($LATEST) der Funktion `my-function` auf 256 MB geändert.

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

Ausgabe:

```
{
```

```
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * Updates the configuration of an AWS Lambda function.
    *
    * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
    * @param functionName  the name of the AWS Lambda function to update
    * @param handler       the new handler for the AWS Lambda function
    *
    * @throws LambdaException if there is an error while updating the function
configuration
    */
   public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
       try {
           UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                   .functionName(functionName)
                   .handler(handler)
                   .runtime(Runtime.JAVA17)
                   .build();

           awsLambda.updateFunctionConfiguration(configurationRequest);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const updateFunctionConfiguration = async (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = await client.send(command);
  await waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function updateFunctionConfiguration($functionName, $handler,
 $environment = '')
    {
```

```
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
        ]);
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für PHP API-Referenz.

PowerShell

Werkzeuge für PowerShell V4

Beispiel 1: Dieses Beispiel aktualisiert die vorhandene Lambda-Funktionskonfiguration

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- API-Details finden Sie unter UpdateFunctionConfiguration in AWS -Tools für PowerShell Cmdlet-Referenz (V4).

Werkzeuge für PowerShell V5

Beispiel 1: Dieses Beispiel aktualisiert die vorhandene Lambda-Funktionskonfiguration

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- API-Details finden Sie unter UpdateFunctionConfiguration in AWS -Tools für PowerShell Cmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS API-Referenz für das SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
                                                     }
                                                   }
                                                 })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
```

```
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;
```

```
        self.wait_for_function_ready().await?;

        Ok(updated)
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für Rust-API-Referenz.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    TRY.
        oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
              iv_functionname = iv_function_name
              iv_runtime = iv_runtime
              iv_description  = 'Updated Lambda function'
              iv_memorysize  = iv_memory_size ).

        MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
```

```
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für SAP ABAP API-Referenz.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Tell the server-side component to log debug output by setting its
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
 async throws {
```

```
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration in AWS SDK für Swift API-Referenz.

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Szenarien für die Verwendung von Lambda AWS SDKs

Die folgenden Codebeispiele zeigen Ihnen, wie Sie gängige Szenarien in Lambda implementieren mit AWS SDKs. Diese Szenarien zeigen Ihnen, wie Sie bestimmte Aufgaben erledigen, indem Sie mehrere Funktionen innerhalb von Lambda oder in Kombination mit anderen AWS-Services aufrufen. Jedes Szenario enthält einen Link zum vollständigen Quell-Code, wo Sie Anleitungen zum Einrichten und Ausführen des Codes finden.

Szenarien zielen auf eine mittlere Erfahrungsebene ab, um Ihnen zu helfen, Service-Aktionen im Kontext zu verstehen.

Beispiele

- Bestätigen Sie automatisch bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion mithilfe einer AWS SDK
- Migrieren Sie automatisch bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion mithilfe einer AWS SDK
- Erstellen Sie eine API Gateway-REST-API zur Nachverfolgung COVID-19 Daten
- Leihbibliothek-REST-API erstellen
- Erstellen einer Messenger-Anwendung mit Step Functions
- Erstellen Sie eine Rest-API mit Funktions-Proxy-Integration
- Eine Anwendung für Foto-Asset-Management erstellen, mit der Benutzer Fotos mithilfe von Labels verwalten können
- Erstellen einer Websocket-Chat-Anwendung mit API Gateway
- Erstellen einer Anwendung, die Kundenfeedback analysiert und Audio generiert
- Erstellen eines Monitoring-Dashboards mit einem Funktionsnamen als Variable
- Erstellen Sie Ihre erste serverlose Funktion
- Aufrufen einer Lambda-Funktion von einem Browser aus
- Transformieren von Daten für Ihre Anwendung mit S3 Object Lambda
- Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion
- Verwenden von Step Functions, um Lambda-Funktionen aufzurufen

- Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion
- Verwenden der Amazon Neptune-API zur Entwicklung einer Lambda-Funktion, die Graphdaten abfragt
- Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen
- Schreiben Sie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito-Benutzerauthentifizierung mithilfe einer AWS SDK

Bestätigen Sie automatisch bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion mithilfe einer AWS SDK

Die folgenden Codebeispiele zeigen, wie bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion automatisch bestätigt werden können.

- Konfigurieren Sie einen Benutzerpool, um eine Lambda-Funktion für den `PreSignUp`-Trigger aufzurufen.
- Registrieren eines Benutzers bei Amazon Cognito.
- Die Lambda-Funktion scannt eine DynamoDB-Tabelle und bestätigt automatisch bekannte Benutzer.
- Melden Sie sich als neuer Benutzer an und bereinigen Sie dann die Ressourcen.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Führen Sie ein interaktives Szenario an einem Prompt aus.

```
import (
 "context"
```

```
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
```

```
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(ctx context.Context, clientId string,
 usersTable string) (string, string) {
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(ctx, usersTable)
 if err != nil {
  panic(err)
 }
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]

 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(ctx, clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("Enter another password:", 8)
   } else {
```

```
    panic(err)
   }
  } else {
   signedUp = true
  }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(ctx context.Context, clientId string,
 userName string, password string) string {
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
 return *authResult.AccessToken
}

// Run runs the scenario.
func (runner *AutoConfirm) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
```

```
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(ctx, stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Behandeln Sie den `PreSignUp`-Trigger mit einer Lambda-Funktion.

```
import (
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"
```

```
// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
```

```
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }

 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Erstellen Sie eine Struktur, die allgemeine Aufgaben ausführt.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
```

```
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
```

```
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Erstellen Sie eine Struktur, die Amazon Cognito-Aktionen umschließt.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
```

```
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
```

```
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
```

```
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
```

```
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
```

```
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Erstellen Sie eine Struktur, die DynamoDB-Aktionen umhüllt.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
```

```
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
```

```
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Erstellen Sie eine Struktur, die umschließt CloudWatch Protokolliert Aktionen.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
```

```
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Erstellen Sie eine Struktur, die umschließt CloudFormation Aktionen.

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
```

```
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Bereinigen von Ressourcen.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
```

```
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
```

```
 }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Konfigurieren Sie einen interaktiven „Szenario“-Lauf. Das JavaScript (v3) -Beispiele teilen sich einen Szenario-Runner, um komplexe Beispiele zu rationalisieren. Der komplette Quellcode ist aktiviert GitHub.

```
import { AutoConfirm } from "./scenario-auto-confirm.js";

/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {
  errors: [],
  users: [
    {
      UserName: "test_user_1",
      UserEmail: "test_email_1@example.com",
    },
```

```
    {
      UserName: "test_user_2",
      UserEmail: "test_email_2@example.com",
    },
    {
      UserName: "test_user_3",
      UserEmail: "test_email_3@example.com",
    },
  ],
};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Demonstrate automatically confirming known users in a database.
  "auto-confirm": AutoConfirm(context),
};

// Call function if run directly
import { fileURLToPath } from "node:url";
import { parseScenarioArgs } from "@aws-doc-sdk-examples/lib/scenario/index.js";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios, {
    name: "Cognito user pools and triggers",
    description:
      "Demonstrate how to use the AWS SDKs to customize Amazon Cognito
 authentication behavior.",
  });
}
```

Dieses Szenario demonstriert die automatische Bestätigung eines bekannten Benutzers. Es orchestriert die Beispielschritte.

```
import { wait } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";
import {
  Scenario,
  ScenarioAction,
  ScenarioInput,
```

```
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";

import {
  getStackOutputs,
  logCleanUpReminder,
  promptForStackName,
  promptForStackRegion,
  skipWhenErrors,
} from "./steps-common.js";
import { populateTable } from "./actions/dynamodb-actions.js";
import {
  addPreSignUpHandler,
  deleteUser,
  getUser,
  signIn,
  signUpUser,
} from "./actions/cognito-actions.js";
import {
  getLatestLogStreamForLambda,
  getLogEvents,
} from "./actions/cloudwatch-logs-actions.js";

/**
 * @typedef {{
 *   errors: Error[],
 *   password: string,
 *   users: { UserName: string, UserEmail: string }[],
 *   selectedUser?: string,
 *   stackName?: string,
 *   stackRegion?: string,
 *   token?: string,
 *   confirmDeleteSignedInUser?: boolean,
 *   TableName?: string,
 *   UserPoolClientId?: string,
 *   UserPoolId?: string,
 *   UserPoolArn?: string,
 *   AutoConfirmHandlerArn?: string,
 *   AutoConfirmHandlerName?: string
 * }} State
 */

const greeting = new ScenarioOutput(
  "greeting",
```

```
  (/** @type {State} */ state) => `This demo will populate some users into the \
database created as part of the "${state.stackName}" stack. \
Then the AutoConfirmHandler will be linked to the PreSignUp \
trigger from Cognito. Finally, you will choose a user to sign up.`,
  { skipWhen: skipWhenErrors },
);

const logPopulatingUsers = new ScenarioOutput(
  "logPopulatingUsers",
  "Populating the DynamoDB table with some users.",
  { skipWhenErrors: skipWhenErrors },
);

const logPopulatingUsersComplete = new ScenarioOutput(
  "logPopulatingUsersComplete",
  "Done populating users.",
  { skipWhen: skipWhenErrors },
);

const populateUsers = new ScenarioAction(
  "populateUsers",
  async (/** @type {State} */ state) => {
    const [_, err] = await populateTable({
      region: state.stackRegion,
      tableName: state.TableName,
      items: state.users,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const logSetupSignUpTrigger = new ScenarioOutput(
  "logSetupSignUpTrigger",
  "Setting up the PreSignUp trigger for the Cognito User Pool.",
  { skipWhen: skipWhenErrors },
);

const setupSignUpTrigger = new ScenarioAction(
  "setupSignUpTrigger",
```

```
  async (/** @type {State} */ state) => {
    const [_, err] = await addPreSignUpHandler({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      handlerArn: state.AutoConfirmHandlerArn,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const logSetupSignUpTriggerComplete = new ScenarioOutput(
  "logSetupSignUpTriggerComplete",
  (
    /** @type {State} */ state,
  ) => `The lambda function "${state.AutoConfirmHandlerName}" \
has been configured as the PreSignUp trigger handler for the user pool
 "${state.UserPoolId}".`,
  { skipWhen: skipWhenErrors },
);

const selectUser = new ScenarioInput(
  "selectedUser",
  "Select a user to sign up.",
  {
    type: "select",
    choices: (/** @type {State} */ state) => state.users.map((u) => u.UserName),
    skipWhen: skipWhenErrors,
    default: (/** @type {State} */ state) => state.users[0].UserName,
  },
);

const checkIfUserAlreadyExists = new ScenarioAction(
  "checkIfUserAlreadyExists",
  async (/** @type {State} */ state) => {
    const [user, err] = await getUser({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      username: state.selectedUser,
    });
```

```
    if (err?.name === "UserNotFoundException") {
      // Do nothing. We're not expecting the user to exist before
      // sign up is complete.
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    if (user) {
      state.errors.push(
        new Error(
          `The user "${state.selectedUser}" already exists in the user pool
 "${state.UserPoolId}".`,
        ),
      );
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const createPassword = new ScenarioInput(
  "password",
  "Enter a password that has at least eight characters, uppercase, lowercase,
 numbers and symbols.",
  { type: "password", skipWhen: skipWhenErrors, default: "Abcd1234!" },
);

const logSignUpExistingUser = new ScenarioOutput(
  "logSignUpExistingUser",
  (/** @type {State} */ state) => `Signing up user "${state.selectedUser}".`,
  { skipWhen: skipWhenErrors },
);

const signUpExistingUser = new ScenarioAction(
  "signUpExistingUser",
  async (/** @type {State} */ state) => {
    const signUp = (password) =>
      signUpUser({
```

```
        region: state.stackRegion,
        userPoolClientId: state.UserPoolClientId,
        username: state.selectedUser,
        email: state.users.find((u) => u.UserName === state.selectedUser)
          .UserEmail,
        password,
      });

    let [_, err] = await signUp(state.password);

    while (err?.name === "InvalidPasswordException") {
      console.warn("The password you entered was invalid.");
      await createPassword.handle(state);
      [_, err] = await signUp(state.password);
    }

    if (err) {
      state.errors.push(err);
    }
  },
  { skipWhen: skipWhenErrors },
);

const logSignUpExistingUserComplete = new ScenarioOutput(
  "logSignUpExistingUserComplete",
  (/** @type {State} */ state) =>
    `"${state.selectedUser} was signed up successfully.`,
  { skipWhen: skipWhenErrors },
);

const logLambdaLogs = new ScenarioAction(
  "logLambdaLogs",
  async (/** @type {State} */ state) => {
    console.log(
      "Waiting a few seconds to let Lambda write to CloudWatch Logs...\n",
    );
    await wait(10);

    const [logStream, logStreamErr] = await getLatestLogStreamForLambda({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
    });
    if (logStreamErr) {
      state.errors.push(logStreamErr);
```

```
      return;
    }

    console.log(
      `Getting some recent events from log stream "${logStream.logStreamName}"`,
    );
    const [logEvents, logEventsErr] = await getLogEvents({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
      eventCount: 10,
      logStreamName: logStream.logStreamName,
    });
    if (logEventsErr) {
      state.errors.push(logEventsErr);
      return;
    }

    console.log(logEvents.map((ev) => `\t${ev.message}`).join(""));
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUser = new ScenarioOutput(
  "logSignInUser",
  (/** @type {State} */ state) => `Let's sign in as ${state.selectedUser}`,
  { skipWhen: skipWhenErrors },
);

const signInUser = new ScenarioAction(
  "signInUser",
  async (/** @type {State} */ state) => {
    const [response, err] = await signIn({
      region: state.stackRegion,
      clientId: state.UserPoolClientId,
      username: state.selectedUser,
      password: state.password,
    });

    if (err?.name === "PasswordResetRequiredException") {
      state.errors.push(new Error("Please reset your password."));
      return;
    }

    if (err) {
```

```
      state.errors.push(err);
      return;
    }

    state.token = response?.AuthenticationResult?.AccessToken;
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUserComplete = new ScenarioOutput(
  "logSignInUserComplete",
  (/** @type {State} */ state) =>
    `Successfully signed in. Your access token starts with:
 ${state.token.slice(0, 11)}`,
  { skipWhen: skipWhenErrors },
);

const confirmDeleteSignedInUser = new ScenarioInput(
  "confirmDeleteSignedInUser",
  "Do you want to delete the currently signed in user?",
  { type: "confirm", skipWhen: skipWhenErrors },
);

const deleteSignedInUser = new ScenarioAction(
  "deleteSignedInUser",
  async (/** @type {State} */ state) => {
    const [_, err] = await deleteUser({
      region: state.stackRegion,
      accessToken: state.token,
    });

    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: (/** @type {State} */ state) =>
      skipWhenErrors(state) || !state.confirmDeleteSignedInUser,
  },
);

const logErrors = new ScenarioOutput(
  "logErrors",
  (/** @type {State}*/ state) => {
```

```
    const errorList = state.errors
      .map((err) => ` - ${err.name}: ${err.message}`)
      .join("\n");
    return `Scenario errors found:\n${errorList}`;
  },
  {
    // Don't log errors when there aren't any!
    skipWhen: (/** @type {State} */ state) => state.errors.length === 0,
  },
);

export const AutoConfirm = (context) =>
  new Scenario(
    "AutoConfirm",
    [
      promptForStackName,
      promptForStackRegion,
      getStackOutputs,
      greeting,
      logPopulatingUsers,
      populateUsers,
      logPopulatingUsersComplete,
      logSetupSignUpTrigger,
      setupSignUpTrigger,
      logSetupSignUpTriggerComplete,
      selectUser,
      checkIfUserAlreadyExists,
      createPassword,
      logSignUpExistingUser,
      signUpExistingUser,
      logSignUpExistingUserComplete,
      logLambdaLogs,
      logSignInUser,
      signInUser,
      logSignInUserComplete,
      confirmDeleteSignedInUser,
      deleteSignedInUser,
      logCleanUpReminder,
      logErrors,
    ],
    context,
  );
```

Dies sind Schritte, die mit anderen Szenarien geteilt werden.

```
import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { getCfnOutputs } from "@aws-doc-sdk-examples/lib/sdk/cfn-outputs.js";

export const skipWhenErrors = (state) => state.errors.length > 0;

export const getStackOutputs = new ScenarioAction(
  "getStackOutputs",
  async (state) => {
    if (!state.stackName || !state.stackRegion) {
      state.errors.push(
        new Error(
          "No stack name or region provided. The stack name and \
region are required to fetch CFN outputs relevant to this example.",
        ),
      );
      return;
    }

    const outputs = await getCfnOutputs(state.stackName, state.stackRegion);
    Object.assign(state, outputs);
  },
);

export const promptForStackName = new ScenarioInput(
  "stackName",
  "Enter the name of the stack you deployed earlier.",
  { type: "input", default: "PoolsAndTriggersStack" },
);

export const promptForStackRegion = new ScenarioInput(
  "stackRegion",
  "Enter the region of the stack you deployed earlier.",
  { type: "input", default: "us-east-1" },
);

export const logCleanUpReminder = new ScenarioOutput(
  "logCleanUpReminder",
  "All done. Remember to run 'cdk destroy' to teardown the stack.",
```

```
  { skipWhen: skipWhenErrors },
);
```

Ein Handler für den `PreSignUp`-Trigger mit einer Lambda-Funktion.

```
import type { PreSignUpTriggerEvent, Handler } from "aws-lambda";
import type { UserRepository } from "./user-repository";
import { DynamoDBUserRepository } from "./user-repository";

export class PreSignUpHandler {
  private userRepository: UserRepository;

  constructor(userRepository: UserRepository) {
    this.userRepository = userRepository;
  }

  private isPreSignUpTriggerSource(event: PreSignUpTriggerEvent): boolean {
    return event.triggerSource === "PreSignUp_SignUp";
  }

  private getEventUserEmail(event: PreSignUpTriggerEvent): string {
    return event.request.userAttributes.email;
  }

  async handlePreSignUpTriggerEvent(
    event: PreSignUpTriggerEvent,
  ): Promise<PreSignUpTriggerEvent> {
    console.log(
      `Received presignup from ${event.triggerSource} for user
 '${event.userName}'`,
    );

    if (!this.isPreSignUpTriggerSource(event)) {
      return event;
    }

    const eventEmail = this.getEventUserEmail(event);
    console.log(`Looking up email ${eventEmail}.`);
    const storedUserInfo =
      await this.userRepository.getUserInfoByEmail(eventEmail);

    if (!storedUserInfo) {
```

```
      console.log(
        `Email ${eventEmail} not found. Email verification is required.`,
      );
      return event;
    }

    if (storedUserInfo.UserName !== event.userName) {
      console.log(
        `UserEmail ${eventEmail} found, but stored UserName
 '${storedUserInfo.UserName}' does not match supplied UserName
 '${event.userName}'. Verification is required.`,
      );
    } else {
      console.log(
        `UserEmail ${eventEmail} found with matching UserName
 ${storedUserInfo.UserName}. User is confirmed.`,
      );
      event.response.autoConfirmUser = true;
      event.response.autoVerifyEmail = true;
    }
    return event;
  }
}

const createPreSignUpHandler = (): PreSignUpHandler => {
  const tableName = process.env.TABLE_NAME;
  if (!tableName) {
    throw new Error("TABLE_NAME environment variable is not set");
  }

  const userRepository = new DynamoDBUserRepository(tableName);
  return new PreSignUpHandler(userRepository);
};

export const handler: Handler = async (event: PreSignUpTriggerEvent) => {
  const preSignUpHandler = createPreSignUpHandler();
  return preSignUpHandler.handlePreSignUpTriggerEvent(event);
};
```

Modul von CloudWatch Protokolliert Aktionen.

```
import {
  CloudWatchLogsClient,
  GetLogEventsCommand,
  OrderBy,
  paginateDescribeLogStreams,
} from "@aws-sdk/client-cloudwatch-logs";

/**
 * Get the latest log stream for a Lambda function.
 * @param {{ functionName: string, region: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").LogStream | null,
 unknown]>}
 */
export const getLatestLogStreamForLambda = async ({ functionName, region }) => {
  try {
    const logGroupName = `/aws/lambda/${functionName}`;
    const cwlClient = new CloudWatchLogsClient({ region });
    const paginator = paginateDescribeLogStreams(
      { client: cwlClient },
      {
        descending: true,
        limit: 1,
        orderBy: OrderBy.LastEventTime,
        logGroupName,
      },
    );

    for await (const page of paginator) {
      return [page.logStreams[0], null];
    }
  } catch (err) {
    return [null, err];
  }
};

/**
 * Get the log events for a Lambda function's log stream.
 * @param {{
 *   functionName: string,
 *   logStreamName: string,
 *   eventCount: number,
 *   region: string
 * }} config
```

```
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").OutputLogEvent[]
 | null, unknown]>}
 */
export const getLogEvents = async ({
  functionName,
  logStreamName,
  eventCount,
  region,
}) => {
  try {
    const cwlClient = new CloudWatchLogsClient({ region });
    const logGroupName = `/aws/lambda/${functionName}`;
    const response = await cwlClient.send(
      new GetLogEventsCommand({
        logStreamName: logStreamName,
        limit: eventCount,
        logGroupName: logGroupName,
      }),
    );

    return [response.events, null];
  } catch (err) {
    return [null, err];
  }
};
```

Modul für Amazon-Cognito-Aktionen.

```
import {
  AdminGetUserCommand,
  CognitoIdentityProviderClient,
  DeleteUserCommand,
  InitiateAuthCommand,
  SignUpCommand,
  UpdateUserPoolCommand,
} from "@aws-sdk/client-cognito-identity-provider";

/**
 * Connect a Lambda function to the PreSignUp trigger for a Cognito user pool
 * @param {{ region: string, userPoolId: string, handlerArn: string }} config
```

```
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").UpdateUserPoolCommandOutput | null, unknown]>}
 */
export const addPreSignUpHandler = async ({
  region,
  userPoolId,
  handlerArn,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const command = new UpdateUserPoolCommand({
      UserPoolId: userPoolId,
      LambdaConfig: {
        PreSignUp: handlerArn,
      },
    });

    const response = await cognitoClient.send(command);
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Attempt to register a user to a user pool with a given username and password.
 * @param {{
 *   region: string,
 *   userPoolClientId: string,
 *   username: string,
 *   email: string,
 *   password: string
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").SignUpCommandOutput | null, unknown]>}
 */
export const signUpUser = async ({
  region,
  userPoolClientId,
  username,
  email,
```

```
  password,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const response = await cognitoClient.send(
      new SignUpCommand({
        ClientId: userPoolClientId,
        Username: username,
        Password: password,
        UserAttributes: [{ Name: "email", Value: email }],
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Sign in a user to Amazon Cognito using a username and password authentication
 flow.
 * @param {{ region: string, clientId: string, username: string, password:
 string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").InitiateAuthCommandOutput | null, unknown]>}
 */
export const signIn = async ({ region, clientId, username, password }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new InitiateAuthCommand({
        AuthFlow: "USER_PASSWORD_AUTH",
        ClientId: clientId,
        AuthParameters: { USERNAME: username, PASSWORD: password },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

```
/**
 * Retrieve an existing user from a user pool.
 * @param {{ region: string, userPoolId: string, username: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").AdminGetUserCommandOutput | null, unknown]>}
 */
export const getUser = async ({ region, userPoolId, username }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new AdminGetUserCommand({
        UserPoolId: userPoolId,
        Username: username,
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Delete the signed-in user. Useful for allowing a user to delete their
 * own profile.
 * @param {{ region: string, accessToken: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").DeleteUserCommandOutput | null, unknown]>}
 */
export const deleteUser = async ({ region, accessToken }) => {
  try {
    const client = new CognitoIdentityProviderClient({ region });
    const response = await client.send(
      new DeleteUserCommand({ AccessToken: accessToken }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

Modul für DynamoDB-Aktionen.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

/**
 * Populate a DynamoDB table with provide items.
 * @param {{ region: string, tableName: string, items: Record<string,
 unknown>[] }} config
 * @returns {Promise<[import("@aws-sdk/lib-dynamodb").BatchWriteCommandOutput |
 null, unknown]>}
 */
export const populateTable = async ({ region, tableName, items }) => {
  try {
    const ddbClient = new DynamoDBClient({ region });
    const docClient = DynamoDBDocumentClient.from(ddbClient);
    const response = await docClient.send(
      new BatchWriteCommand({
        RequestItems: {
          [tableName]: items.map((item) => ({
            PutRequest: {
              Item: item,
            },
          })),
        },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für JavaScript -API-Referenz.
  - DeleteUser
  - InitiateAuth
  - SignUp

- UpdateUserPool

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Migrieren Sie automatisch bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion mithilfe einer AWS SDK

Das folgende Codebeispiel zeigt, wie bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion automatisch migriert werden können.

- Konfigurieren Sie einen Benutzerpool, um eine Lambda-Funktion für den `MigrateUser`-Trigger aufzurufen.
- Melden Sie sich bei Amazon Cognito mit einem Benutzernamen und einer E-Mail-Adresse an, die sich nicht im Benutzerpool befinden.
- Die Lambda-Funktion durchsucht eine DynamoDB-Tabelle und migriert bekannte Benutzer automatisch in den Benutzerpool.
- Führen Sie den Ablauf „Passwort vergessen“ aus, um das Passwort für den migrierten Benutzer zurückzusetzen.
- Melden Sie sich als neuer Benutzer an und bereinigen Sie dann die Ressourcen.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Führen Sie ein interaktives Szenario an einem Prompt aus.

```
import (
 "context"
```

```
 "errors"
 "fmt"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
func (runner *MigrateUser) AddMigrateUserTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
```

```
  "rejects the user.\n\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(ctx context.Context, usersTable string,
 clientId string) (bool, actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")

 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")

 runner.helper.AddKnownUser(ctx, usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(ctx, clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
```

```
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
   runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
   signedIn = true
  }
 }

 log.Println(strings.Repeat("-", 88))
 return resetRequired != nil, user
}

// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(ctx context.Context, clientId string,
 user actions.User) {
 wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
 to Cognito, you must be able to receive a confirmation\n"+
  "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
 if !wantCode {
  log.Println("To complete this example and successfully migrate a user to
 Cognito, you must enter an email\n" +
   "you own that can receive a confirmation code.")
  return
 }
 codeDelivery, err := runner.cognitoActor.ForgotPassword(ctx, clientId,
 user.UserName)
 if err != nil {
  panic(err)
 }
 log.Printf("\nA confirmation code has been sent to %v.",
 *codeDelivery.Destination)
 code := runner.questioner.Ask("Check your email and enter it here:")
```

```
 confirmed := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !confirmed {
  log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.ConfirmForgotPassword(ctx, clientId, code,
 user.UserName, password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   confirmed = true
  }
 }
 log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
 log.Println("Signing in with your username and password...")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, user.UserName,
 password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)

 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *MigrateUser) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
```

```
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(ctx, stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(ctx, stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(ctx, stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Behandeln Sie den `MigrateUser`-Trigger mit einer Lambda-Funktion.

```
import (
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
```

```
  ExpressionAttributeValues: expr.Values(),
 })
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }

 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
```

```
}
```

Erstellen Sie eine Struktur, die allgemeine Aufgaben ausführt.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}
```

```
// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
```

```
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Erstellen Sie eine Struktur, die Amazon Cognito-Aktionen umschließt.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
```

```
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
```

```
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}
```

```
// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
```

```
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
```

```
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Erstellen Sie eine Struktur, die DynamoDB-Aktionen umhüllt.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}
```

```
// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
```

```
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
```

```
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Erstellen Sie eine Struktur, die umschließt CloudWatch Protokolliert Aktionen.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
```

```
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Erstellen Sie eine Struktur, die umschließt CloudFormation Aktionen.

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}
```

```
// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Bereinigen von Ressourcen.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}
```

```
func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
```

```
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - ConfirmForgotPassword
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUserPool

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen Sie eine API Gateway-REST-API zur Nachverfolgung COVID-19 Daten

Das folgende Codebeispiel zeigt, wie Sie eine REST-API erstellen, die ein System simuliert, um tägliche Fälle von COVID-19 in den Vereinigten Staaten unter Verwendung fiktiver Daten.

Python

SDK für Python (Boto3)

Zeigt, wie man es benutzt AWS Kelch mit dem AWS SDK für Python (Boto3) um eine serverlose REST-API zu erstellen, die Amazon API Gateway verwendet, AWS Lambda, und Amazon DynamoDB. Die REST-API simuliert ein System, das tägliche Fälle von verfolgt COVID-19 in den Vereinigten Staaten unter Verwendung fiktiver Daten. Lernen Sie Folgendes:

- Verwenden AWS Chalice, um Routen in Lambda-Funktionen zu definieren, die aufgerufen werden, um REST-Anfragen zu bearbeiten, die über das API Gateway eingehen.

- Verwenden Sie Lambda-Funktionen zum Abrufen und Speichern von Daten in einer DynamoDB-Tabelle, um REST-Anforderungen zu bearbeiten.
- Definieren Sie Ressourcen für Tabellenstruktur und Sicherheitsrolle in einer AWS CloudFormation Vorlage.
- Verwenden AWS Kelch und CloudFormation um alle notwendigen Ressourcen zu verpacken und bereitzustellen.
- Verwenden CloudFormation um alle erstellten Ressourcen zu bereinigen.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- CloudFormation
- DynamoDB
- Lambda

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Leihbibliothek-REST-API erstellen

Im folgenden Codebeispiel wird veranschaulicht, wie man eine Leihbibliothek erstellt, in der Kunden Bücher mithilfe einer REST-API ausleihen und zurückgeben können, die von einer Amazon-Aurora-Datenbank unterstützt wird.

Python

SDK für Python (Boto3)

Zeigt, wie man das benutzt AWS SDK für Python (Boto3) mit der Amazon Relational Database Service (Amazon RDS) API und AWS Chalice, um eine REST-API zu erstellen, die von einer Amazon Aurora-Datenbank unterstützt wird. Der Webservice ist vollständig Serverless und stellt eine einfache Leihbibliothek dar, in der die Kunden Bücher ausleihen und zurückgeben können. Lernen Sie Folgendes:

- Erstellen und verwalten Sie einen Serverless-Aurora-Datenbank-Cluster.

- Verwenden Sie AWS Secrets Manager um Datenbankanmeldeinformationen zu verwalten.
- Implementieren Sie einen Datenspeicher-Layer, der Amazon RDS verwendet, um Daten in die und aus der Datenbank zu verschieben.
- Verwenden AWS Chalice, um eine serverlose REST-API für Amazon API Gateway bereitzustellen und AWS Lambda.
- Verwenden Sie das Anforderungspaket, um Anfragen an den Webservice zu senden.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- Aurora
- Lambda
- Secrets Manager

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer Messenger-Anwendung mit Step Functions

Das folgende Codebeispiel zeigt, wie Sie ein erstellen AWS Step Functions Messenger-Anwendung, die Nachrichtendatensätze aus einer Datenbanktabelle abruft.

Python

SDK für Python (Boto3)

Zeigt, wie man das benutzt AWS SDK für Python (Boto3) mit AWS Step Functions um eine Messenger-Anwendung zu erstellen, die Nachrichtendatensätze aus einer Amazon DynamoDB-Tabelle abruft und sie mit Amazon Simple Queue Service (Amazon SQS) sendet. Die Zustandsmaschine integriert sich in eine AWS Lambda Funktion zum Durchsuchen der Datenbank nach noch nicht gesendeten Nachrichten.

- Erstellen Sie einen Zustandsautomaten, der Nachrichtendatensätze aus einer Amazon-DynamoDB-Tabelle abruft und aktualisiert.

- Aktualisieren Sie die Definition des Zustandsautomaten, um auch Nachrichten an Amazon Simple Queue Service (Amazon SQS) zu senden.
- Starten und stoppen Sie Ausführungen des Zustandsautomaten.
- Stellen Sie vom Zustandsautomaten aus über Serviceintegrationen eine Verbindung zu Lambda, DynamoDB und Amazon SQS her.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB
- Lambda
- Amazon SQS
- Step-Funktionen

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen Sie eine Rest-API mit Funktions-Proxy-Integration

Wie das aussehen kann, sehen Sie am nachfolgenden Beispielcode:

- Erstellen Sie eine IAM-Rolle für die Lambda-Ausführung
- Erstellen Sie eine Lambda-Funktion und stellen Sie sie bereit
- Erstellen Sie eine REST-API
- Konfigurieren Sie die Lambda-Proxy-Integration
- Bereitstellen und Testen der API
- Bereinigen von Ressourcen

Bash

AWS CLI mit Bash-Skript

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie es im Tutorials und Beispiele-Repository für Entwickler einrichten und ausführen.

```
#!/bin/bash

set -euo pipefail

# Simple API Gateway Lambda Integration Script
# This script creates a REST API with Lambda proxy integration

# Generate random identifiers
FUNCTION_NAME="GetStartedLambdaProxyIntegration-$(openssl rand -hex 4)"
ROLE_NAME="GetStartedLambdaBasicExecutionRole-$(openssl rand -hex 4)"
API_NAME="LambdaProxyAPI-$(openssl rand -hex 4)"

# Get AWS account info
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=$(aws configure get region || echo "us-east-1")

# Validate inputs
if [[ -z "$ACCOUNT_ID" ]] || [[ -z "$REGION" ]]; then
    echo "Error: Failed to retrieve AWS account information" >&2
    exit 1
fi

echo "Creating Lambda function code..."

# Create Lambda function code with input validation
cat > lambda_function.py << 'EOF'
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
def lambda_handler(event, context):
    try:
        logger.info("Received event: %s", json.dumps(event))

        greeter = 'World'

        # Safely retrieve greeter from query string parameters
        query_params = event.get('queryStringParameters') or {}
        if isinstance(query_params, dict) and 'greeter' in query_params:
            greeter_value = query_params.get('greeter')
            if isinstance(greeter_value, str) and greeter_value:
                greeter = greeter_value

        # Safely retrieve greeter from multi-value headers
        multi_headers = event.get('multiValueHeaders') or {}
        if isinstance(multi_headers, dict) and 'greeter' in multi_headers:
            greeter_list = multi_headers.get('greeter', [])
            if isinstance(greeter_list, list) and greeter_list:
                greeter = " and ".join(str(g) for g in greeter_list if g)

        # Safely retrieve greeter from headers
        headers = event.get('headers') or {}
        if isinstance(headers, dict) and 'greeter' in headers:
            greeter_value = headers.get('greeter')
            if isinstance(greeter_value, str) and greeter_value:
                greeter = greeter_value

        # Safely retrieve greeter from body
        body = event.get('body')
        if body and isinstance(body, str):
            try:
                body_dict = json.loads(body)
                if isinstance(body_dict, dict) and 'greeter' in body_dict:
                    greeter_value = body_dict.get('greeter')
                    if isinstance(greeter_value, str) and greeter_value:
                        greeter = greeter_value
            except (json.JSONDecodeError, ValueError) as e:
                logger.warning("Failed to parse body: %s", str(e))

        # Sanitize greeter to prevent injection
        greeter = greeter.replace('"', '\\"').replace("'", "\\'")

        response = {
            "statusCode": 200,
```

```
            "headers": {
                "Content-Type": "application/json"
            },
            "body": json.dumps({"message": f"Hello, {greeter}!"})
        }

        logger.info("Response: %s", json.dumps(response))
        return response

    except Exception as e:
        logger.error("Unexpected error: %s", str(e), exc_info=True)
        return {
            "statusCode": 500,
            "headers": {
                "Content-Type": "application/json"
            },
            "body": json.dumps({"error": "Internal server error"})
        }
EOF

# Create deployment package
zip -q function.zip lambda_function.py || {
    echo "Error: Failed to create function.zip" >&2
    exit 1
}

echo "Creating IAM role..."

# Create IAM trust policy
cat > trust-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

```
# Create IAM role with error handling
aws iam create-role \
    --role-name "$ROLE_NAME" \
    --assume-role-policy-document file://trust-policy.json \
    --description "Temporary role for Lambda execution" || {
    echo "Error: Failed to create IAM role" >&2
    exit 1
}

aws iam tag-role --role-name "$ROLE_NAME" --tags Key=project,Value=doc-smith
 Key=tutorial,Value=apigateway-lambda-integration

# Attach execution policy
aws iam attach-role-policy \
    --role-name "$ROLE_NAME" \
    --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" || {
    echo "Error: Failed to attach IAM policy" >&2
    exit 1
}

# Wait for role propagation
sleep 15

echo "Creating Lambda function..."

# Create Lambda function with Python 3.11 (more recent runtime)
aws lambda create-function \
    --function-name "$FUNCTION_NAME" \
    --runtime python3.11 \
    --role "arn:aws:iam::$ACCOUNT_ID:role/$ROLE_NAME" \
    --handler lambda_function.lambda_handler \
    --zip-file fileb://function.zip \
    --timeout 30 \
    --memory-size 128 \
    --environment "Variables={LOG_LEVEL=INFO}" \
    --tags project=doc-smith,tutorial=apigateway-lambda-integration || {
    echo "Error: Failed to create Lambda function" >&2
    exit 1
}

echo "Creating API Gateway..."

# Create REST API with minimum logging
```

```
API_RESPONSE=$(aws apigateway create-rest-api \
    --name "$API_NAME" \
    --endpoint-configuration types=REGIONAL \
    --description "API for Lambda proxy integration tutorial" \
    --tags project=doc-smith,tutorial=apigateway-lambda-integration \
    --output json)

API_ID=$(echo "$API_RESPONSE" | grep -o '"id": "[^"]*"' | head -1 | cut -d'"' -
f4)

if [[ -z "$API_ID" ]]; then
    echo "Error: Failed to create API Gateway" >&2
    exit 1
fi

# Get root resource ID
ROOT_RESOURCE_ID=$(aws apigateway get-resources --rest-api-id "$API_ID" --query
 'items[?path==`/`].id' --output text)

# Create helloworld resource
aws apigateway create-resource \
    --rest-api-id "$API_ID" \
    --parent-id "$ROOT_RESOURCE_ID" \
    --path-part helloworld || {
    echo "Error: Failed to create resource" >&2
    exit 1
}

# Get resource ID
RESOURCE_ID=$(aws apigateway get-resources --rest-api-id "$API_ID" --query
 "items[?pathPart=='helloworld'].id" --output text)

# Create ANY method with no authorization (intentional for tutorial)
aws apigateway put-method \
    --rest-api-id "$API_ID" \
    --resource-id "$RESOURCE_ID" \
    --http-method ANY \
    --authorization-type NONE || {
    echo "Error: Failed to create method" >&2
    exit 1
}

# Set up Lambda proxy integration
```

```
LAMBDA_URI="arn:aws:apigateway:$REGION:lambda:path/2015-03-31/functions/
arn:aws:lambda:$REGION:$ACCOUNT_ID:function:$FUNCTION_NAME/invocations"

aws apigateway put-integration \
    --rest-api-id "$API_ID" \
    --resource-id "$RESOURCE_ID" \
    --http-method ANY \
    --type AWS_PROXY \
    --integration-http-method POST \
    --uri "$LAMBDA_URI" || {
    echo "Error: Failed to create integration" >&2
    exit 1
}

# Grant API Gateway permission to invoke Lambda
STATEMENT_ID="apigateway-invoke-$(openssl rand -hex 4)"
SOURCE_ARN="arn:aws:execute-api:$REGION:$ACCOUNT_ID:$API_ID/*/*"

aws lambda add-permission \
    --function-name "$FUNCTION_NAME" \
    --statement-id "$STATEMENT_ID" \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "$SOURCE_ARN" || {
    echo "Error: Failed to add Lambda permission" >&2
    exit 1
}

# Deploy API
aws apigateway create-deployment \
    --rest-api-id "$API_ID" \
    --stage-name test \
    --description "Test deployment" || {
    echo "Error: Failed to deploy API" >&2
    exit 1
}

echo "Testing API..."

# Test the API
INVOKE_URL="https://$API_ID.execute-api.$REGION.amazonaws.com/test/helloworld"

echo "API URL: $INVOKE_URL"
```

```
# Test with query parameter (with proper URL encoding)
echo "Testing with query parameter:"
curl -s -X GET "$INVOKE_URL?greeter=John" | jq . 2>/dev/null || curl -s -X GET
 "$INVOKE_URL?greeter=John"
echo ""

# Test with header
echo "Testing with header:"
curl -s -X GET "$INVOKE_URL" \
    -H 'content-type: application/json' \
    -H 'greeter: John' | jq . 2>/dev/null || curl -s -X GET "$INVOKE_URL" \
    -H 'content-type: application/json' \
    -H 'greeter: John'
echo ""

# Test with body
echo "Testing with POST body:"
curl -s -X POST "$INVOKE_URL" \
    -H 'content-type: application/json' \
    -d '{"greeter": "John"}' | jq . 2>/dev/null || curl -s -X POST "$INVOKE_URL"
 \
    -H 'content-type: application/json' \
    -d '{"greeter": "John"}'
echo ""

echo "Tutorial completed! API is available at: $INVOKE_URL"

# Cleanup
echo "Cleaning up resources..."

# Delete API
aws apigateway delete-rest-api --rest-api-id "$API_ID" || echo "Warning: Failed
 to delete API" >&2

# Delete Lambda function
aws lambda delete-function --function-name "$FUNCTION_NAME" || echo "Warning:
 Failed to delete Lambda function" >&2

# Detach policy and delete role
aws iam detach-role-policy \
    --role-name "$ROLE_NAME" \
    --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" || echo "Warning: Failed to detach policy" >&2
```

```
aws iam delete-role --role-name "$ROLE_NAME" || echo "Warning: Failed to delete
 role" >&2

# Clean up local files securely
rm -f lambda_function.py function.zip trust-policy.json

echo "Cleanup completed!"
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS CLI - Befehlsreferenz.
  - AddPermission
  - AttachRolePolicy
  - CreateDeployment
  - CreateFunction
  - CreateResource
  - CreateRestApi
  - CreateRole
  - DeleteFunction
  - DeleteRestApi
  - DeleteRole
  - DetachRolePolicy
  - GetCallerIdentity
  - GetResources
  - GetRestApis
  - PutIntegration
  - PutMethod

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Eine Anwendung für Foto-Asset-Management erstellen, mit der Benutzer Fotos mithilfe von Labels verwalten können

Die folgenden Codebeispiele zeigen, wie eine Serverless-Anwendung erstellt wird, mit der Benutzer Fotos mithilfe von Labels verwalten können.

.NET

SDK für .NET

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK für C++

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK für Java 2.x

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK für JavaScript (v3)

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK für Kotlin

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK für PHP

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK für Rust

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

- Amazon Rekognition
- Amazon S3
- Amazon SNS

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer Websocket-Chat-Anwendung mit API Gateway

Das folgende Codebeispiel zeigt, wie eine Chat-Anwendung erstellt wird, die von einer auf Amazon API Gateway basierenden Websocket-API bereitgestellt wird.

Python

SDK für Python (Boto3)

Zeigt, wie man das benutzt AWS SDK für Python (Boto3) mit Amazon API Gateway V2, um eine Websocket-API zu erstellen, die sich integrieren lässt AWS Lambda und Amazon DynamoDB.

- Erstellen Sie eine WebSocket-API, die von API Gateway bereitgestellt wird.
- Definieren Sie einen Lambda-Handler, der Verbindungen in DynamoDB speichert und Nachrichten an andere Chat-Teilnehmer sendet.
- Stellen Sie eine Verbindung zur Websocket-Chat-Anwendung her und senden Sie Nachrichten mit dem Websockets-Paket.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer Anwendung, die Kundenfeedback analysiert und Audio generiert

Die folgenden Codebeispiele zeigen, wie Sie eine Anwendung erstellen, die Kundenkommentarkarten analysiert, sie aus ihrer Originalsprache übersetzt, ihre Stimmung ermittelt und aus dem übersetzten Text eine Audiodatei generiert.

.NET

SDK für .NET

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen. Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub.

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK für Java 2.x

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält

Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen. Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub.

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK für JavaScript (v3)

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen. Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub. Die folgenden Auszüge zeigen, wie AWS SDK für JavaScript wird innerhalb von Lambda-Funktionen verwendet.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";
```

```
/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );

  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
```

```
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});
```

```
  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK für Ruby

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen. Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub.

In diesem Beispiel verwendete Dienste

- Amazon Comprehend

- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen eines Monitoring-Dashboards mit einem Funktionsnamen als Variable

Wie das aussehen kann, sehen Sie am nachfolgenden Beispielcode:

- Erstellen Sie ein CloudWatch Dashboard
- Fügen Sie Lambda-Metrik-Widgets mit einer Funktionsnamenvariablen hinzu
- Überprüfen Sie das Dashboard
- Bereinigen von Ressourcen

Bash

AWS CLI mit Bash-Skript

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie es im Tutorials und Beispiele-Repository für Entwickler einrichten und ausführen.

```
#!/bin/bash

# Script to create a CloudWatch dashboard with Lambda function name as a variable
# This script creates a CloudWatch dashboard that allows you to switch between
 different Lambda functions

# Set up logging with secure permissions
LOG_FILE="${HOME}/.cloudwatch-dashboard-script.log"
```

```
touch "$LOG_FILE" && chmod 600 "$LOG_FILE"
exec > >(tee -a "$LOG_FILE") 2>&1

echo "$(date): Starting CloudWatch dashboard creation script"

# Security: Set strict error handling
set -uo pipefail
trap 'handle_error "Script failed at line $LINENO"' ERR

# Function to handle errors
handle_error() {
    local error_msg="${1:-Unknown error}"
    echo "ERROR: $error_msg" >&2
    echo "Resources created:"
    echo "- CloudWatch Dashboard: LambdaMetricsDashboard"
    echo ""
    echo "=========================================="
    echo "CLEANUP CONFIRMATION"
    echo "=========================================="
    echo "An error occurred. Proceeding with automatic cleanup..."

    echo "Cleaning up resources..."
    aws cloudwatch delete-dashboards --dashboard-names LambdaMetricsDashboard 2>/
dev/null || true

    # Clean up temporary files
    if [ -n "${TEMP_DIR:-}" ] && [ -d "$TEMP_DIR" ]; then
        rm -rf "$TEMP_DIR"
    fi
    rm -f dashboard-body.json

    echo "Cleanup complete."
    exit 1
}

# Security: Validate AWS CLI is installed
if ! command -v aws &> /dev/null; then
    handle_error "AWS CLI is not installed. Please install it and try again."
fi

# Check if AWS CLI is installed and configured
echo "Checking AWS CLI configuration..."
if ! aws sts get-caller-identity > /dev/null 2>&1; then
```

```
    handle_error "AWS CLI is not properly configured. Please configure it with
 'aws configure' and try again."
fi

# Get the current region securely
REGION=$(aws configure get region 2>/dev/null || echo "")
if [ -z "$REGION" ]; then
    REGION="us-east-1"
    echo "No region found in AWS config, defaulting to $REGION"
fi
echo "Using region: $REGION"

# Validate region format
if ! [[ "$REGION" =~ ^[a-z]{2}-[a-z]+-[0-9]{1}$ ]]; then
    handle_error "Invalid AWS region format: $REGION"
fi

# Check if there are any Lambda functions in the account
echo "Checking for Lambda functions..."
LAMBDA_FUNCTIONS=$(aws lambda list-functions --region "$REGION" --query
 "Functions[*].FunctionName" --output text 2>/dev/null || echo "")

if [ -z "$LAMBDA_FUNCTIONS" ]; then
    echo "No Lambda functions found in your account. Creating a simple test
 function..."

    # Create a temporary directory for Lambda function code with secure
 permissions
    TEMP_DIR=$(mktemp -d)
    chmod 700 "$TEMP_DIR"
    trap 'rm -rf "$TEMP_DIR"' EXIT

    # Create a simple Lambda function
    cat > "$TEMP_DIR/index.js" << 'EOF'
exports.handler = async (event) => {
    console.log('Event:', JSON.stringify(event, null, 2));
    return {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
};
EOF

    # Zip the function code
```

```
    if ! cd "$TEMP_DIR"; then
        handle_error "Failed to change to temporary directory"
    fi

    if ! zip -q function.zip index.js; then
        handle_error "Failed to create zip file"
    fi

    # Create a role for the Lambda function with restricted trust policy
    ROLE_NAME="LambdaDashboardTestRole-$(date +%s)"
    TRUST_POLICY='{"Version":"2012-10-17",       "Statement":
[{"Effect":"Allow","Principal":
{"Service":"lambda.amazonaws.com"},"Action":"sts:AssumeRole"}]}'

    if ! ROLE_ARN=$(aws iam create-role \
        --role-name "$ROLE_NAME" \
        --assume-role-policy-document "$TRUST_POLICY" \
        --tags Key=project,Value=doc-smith Key=tutorial,Value=cloudwatch-
dynamicdash \
        --query "Role.Arn" \
        --output text 2>/dev/null); then
        handle_error "Failed to create IAM role for Lambda function"
    fi

    echo "Waiting for role to be available..."
    sleep 10

    # Attach basic Lambda execution policy
    if ! aws iam attach-role-policy \
        --role-name "$ROLE_NAME" \
        --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"; then
        aws iam delete-role --role-name "$ROLE_NAME" 2>/dev/null || true
        handle_error "Failed to attach policy to IAM role"
    fi

    # Create the Lambda function
    FUNCTION_NAME="DashboardTestFunction-$(date +%s)"
    if ! aws lambda create-function \
        --function-name "$FUNCTION_NAME" \
        --runtime nodejs18.x \
        --role "$ROLE_ARN" \
        --handler index.handler \
        --zip-file fileb://function.zip \
```

```
        --tags project=doc-smith,tutorial=cloudwatch-dynamicdash \
        --region "$REGION" > /dev/null 2>&1; then
        aws iam detach-role-policy \
            --role-name "$ROLE_NAME" \
            --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" 2>/dev/null || true
        aws iam delete-role --role-name "$ROLE_NAME" 2>/dev/null || true
        handle_error "Failed to create Lambda function"
    fi

    # Invoke the function to generate some metrics
    echo "Invoking Lambda function to generate metrics..."
    for i in {1..5}; do
        aws lambda invoke --function-name "$FUNCTION_NAME" --payload '{}' /dev/
null --region "$REGION" > /dev/null 2>&1 || true
        sleep 1
    done

    # Go back to original directory
    cd - > /dev/null

    # Set the function name for the dashboard
    DEFAULT_FUNCTION="$FUNCTION_NAME"
else
    # Use the first Lambda function as default
    DEFAULT_FUNCTION=$(echo "$LAMBDA_FUNCTIONS" | awk '{print $1}')
    echo "Found Lambda functions. Using $DEFAULT_FUNCTION as default."
    FUNCTION_NAME=""
    ROLE_NAME=""
fi

# Create a dashboard with Lambda metrics and a function name variable
echo "Creating CloudWatch dashboard with Lambda function name variable..."

# Create a JSON file for the dashboard body with secure permissions
DASHBOARD_JSON="dashboard-body-$$.json"
touch "$DASHBOARD_JSON" && chmod 600 "$DASHBOARD_JSON"

# Escape special characters in region and function name for JSON
REGION_ESCAPED=$(printf '%s\n' "$REGION" | sed 's:[\/&]:\\&:g')
FUNCTION_ESCAPED=$(printf '%s\n' "$DEFAULT_FUNCTION" | sed 's:[\/&]:\\&:g')

cat > "$DASHBOARD_JSON" << EOF
{
```

```
 "widgets": [
   {
     "type": "metric",
     "x": 0,
     "y": 0,
     "width": 12,
     "height": 6,
     "properties": {
       "metrics": [
         [ "AWS/Lambda", "Invocations", "FunctionName", "\${FunctionName}" ],
         [ ".", "Errors", ".", "." ],
         [ ".", "Throttles", ".", "." ]
       ],
       "view": "timeSeries",
       "stacked": false,
       "region": "$REGION_ESCAPED",
       "title": "Lambda Function Metrics for \${FunctionName}",
       "period": 300
     }
   },
   {
     "type": "metric",
     "x": 0,
     "y": 6,
     "width": 12,
     "height": 6,
     "properties": {
       "metrics": [
         [ "AWS/Lambda", "Duration", "FunctionName", "\${FunctionName}",
{ "stat": "Average" } ]
       ],
       "view": "timeSeries",
       "stacked": false,
       "region": "$REGION_ESCAPED",
       "title": "Duration for \${FunctionName}",
       "period": 300
     }
   },
   {
     "type": "metric",
     "x": 12,
     "y": 0,
     "width": 12,
     "height": 6,
```

```
      "properties": {
        "metrics": [
          [ "AWS/Lambda", "ConcurrentExecutions", "FunctionName", "\
${FunctionName}" ]
        ],
        "view": "timeSeries",
        "stacked": false,
        "region": "$REGION_ESCAPED",
        "title": "Concurrent Executions for \${FunctionName}",
        "period": 300
      }
    }
  ],
  "periodOverride": "auto",
  "variables": [
    {
      "type": "property",
      "id": "FunctionName",
      "property": "FunctionName",
      "label": "Lambda Function",
      "inputType": "select",
      "values": [
        {
          "value": "$FUNCTION_ESCAPED",
          "label": "$FUNCTION_ESCAPED"
        }
      ]
    }
  ]
}
EOF

# Validate JSON before sending
if ! jq empty "$DASHBOARD_JSON" 2>/dev/null; then
    handle_error "Invalid JSON generated for dashboard"
fi

# Create the dashboard using the JSON file
DASHBOARD_NAME="LambdaMetricsDashboard-$(date +%s)"
if ! DASHBOARD_RESULT=$(aws cloudwatch put-dashboard \
    --dashboard-name "$DASHBOARD_NAME" \
    --dashboard-body file://"$DASHBOARD_JSON" \
    --region "$REGION" 2>&1); then
    # If we created resources, clean them up
```

```
    if [ -n "${FUNCTION_NAME:-}" ]; then
        aws lambda delete-function --function-name "$FUNCTION_NAME" --region
 "$REGION" 2>/dev/null || true
        aws iam detach-role-policy \
            --role-name "$ROLE_NAME" \
            --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" 2>/dev/null || true
        aws iam delete-role --role-name "$ROLE_NAME" 2>/dev/null || true
    fi
    handle_error "Failed to create CloudWatch dashboard."
fi

# Display any validation messages but continue
if echo "$DASHBOARD_RESULT" | grep -q "DashboardValidationMessages"; then
    echo "Dashboard created with validation messages:"
    echo "$DASHBOARD_RESULT"
    echo "These validation messages are warnings and the dashboard should still
 function."
else
    echo "Dashboard created successfully!"
fi

# Verify the dashboard was created
echo "Verifying dashboard creation..."
if ! DASHBOARD_INFO=$(aws cloudwatch get-dashboard --dashboard-name
 "$DASHBOARD_NAME" --region "$REGION" 2>&1); then
    # If we created resources, clean them up
    if [ -n "${FUNCTION_NAME:-}" ]; then
        aws lambda delete-function --function-name "$FUNCTION_NAME" --region
 "$REGION" 2>/dev/null || true
        aws iam detach-role-policy \
            --role-name "$ROLE_NAME" \
            --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" 2>/dev/null || true
        aws iam delete-role --role-name "$ROLE_NAME" 2>/dev/null || true
    fi
    handle_error "Failed to verify dashboard creation."
fi

echo "Dashboard verification successful!"
echo "Dashboard details:"
echo "$DASHBOARD_INFO"

# List all dashboards to confirm
```

```
echo "Listing all dashboards:"
if ! DASHBOARDS=$(aws cloudwatch list-dashboards --region "$REGION" 2>&1); then
    # If we created resources, clean them up
    if [ -n "${FUNCTION_NAME:-}" ]; then
        aws lambda delete-function --function-name "$FUNCTION_NAME" --region
 "$REGION" 2>/dev/null || true
        aws iam detach-role-policy \
            --role-name "$ROLE_NAME" \
            --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" 2>/dev/null || true
        aws iam delete-role --role-name "$ROLE_NAME" 2>/dev/null || true
    fi
    handle_error "Failed to list dashboards."
fi
echo "$DASHBOARDS"

# Show instructions for accessing the dashboard
echo ""
echo "Dashboard created successfully! To access it:"
echo "1. Open the CloudWatch console at https://console.aws.amazon.com/
cloudwatch/"
echo "2. In the navigation pane, choose Dashboards"
echo "3. Select $DASHBOARD_NAME"
echo "4. You should see a dropdown menu labeled 'Lambda Function' at the top of
 the dashboard"
echo "5. Use this dropdown to select different Lambda functions and see their
 metrics"
echo ""

# Create a list of resources for cleanup
RESOURCES=("- CloudWatch Dashboard: $DASHBOARD_NAME")
if [ -n "${FUNCTION_NAME:-}" ]; then
    RESOURCES+=("- Lambda Function: $FUNCTION_NAME")
    RESOURCES+=("- IAM Role: $ROLE_NAME")
fi

# Prompt for cleanup with automatic yes
echo "==========================================="
echo "CLEANUP CONFIRMATION"
echo "==========================================="
echo "Resources created:"
for resource in "${RESOURCES[@]}"; do
    echo "$resource"
done
```

```
echo ""
echo "Proceeding with automatic cleanup..."

CLEANUP_CHOICE="y"

if [[ "${CLEANUP_CHOICE,,}" == "y" ]]; then
    echo "Cleaning up resources..."

    # Delete the dashboard
    if aws cloudwatch delete-dashboards --dashboard-names "$DASHBOARD_NAME" --
region "$REGION" 2>/dev/null; then
        echo "Dashboard deleted successfully."
    else
        echo "WARNING: Failed to delete dashboard. You may need to delete it
 manually."
    fi

    # If we created a Lambda function, delete it and its role
    if [ -n "${FUNCTION_NAME:-}" ]; then
        echo "Deleting Lambda function..."
        if aws lambda delete-function --function-name "$FUNCTION_NAME" --region
 "$REGION" 2>/dev/null; then
            echo "Lambda function deleted successfully."
        else
            echo "WARNING: Failed to delete Lambda function. You may need to
 delete it manually."
        fi

        echo "Detaching role policy..."
        if aws iam detach-role-policy \
            --role-name "$ROLE_NAME" \
            --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" 2>/dev/null; then
            echo "Role policy detached successfully."
        else
            echo "WARNING: Failed to detach role policy. You may need to detach
 it manually."
        fi

        echo "Deleting IAM role..."
        if aws iam delete-role --role-name "$ROLE_NAME" 2>/dev/null; then
            echo "IAM role deleted successfully."
        else
```

```
            echo "WARNING: Failed to delete IAM role. You may need to delete it
 manually."
        fi
    fi

    # Clean up the JSON file
    rm -f "$DASHBOARD_JSON"

    echo "Cleanup complete."
fi

echo "Script completed successfully!"
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS CLI - Befehlsreferenz.
  - AttachRolePolicy
  - CreateFunction
  - CreateRole
  - DeleteDashboards
  - DeleteFunction
  - DeleteRole
  - DetachRolePolicy
  - GetCallerIdentity
  - GetDashboard
  - Invoke
  - ListDashboards
  - ListFunctions
  - PutDashboard

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen Sie Ihre erste serverlose Funktion

Wie das aussehen kann, sehen Sie am nachfolgenden Beispielcode:

- Erstellen Sie eine IAM-Rolle für Lambda
- Funktionscode erstellen
- Erstellen einer Lambda-Funktion
- Testen Sie Ihre Lambda-Funktion
- Bereinigen von Ressourcen

Bash

AWS CLI mit Bash-Skript

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie es im Tutorials und Beispiele-Repository für Entwickler einrichten und ausführen.

```
#!/bin/bash
# AWS Lambda - Create Your First Function
# This script creates a Lambda function, invokes it with a test event,
# views CloudWatch logs, and cleans up all resources.
#
# Source: https://docs.aws.amazon.com/lambda/latest/dg/getting-started.html
#
# Resources created:
#   - IAM role (Lambda execution role with basic logging permissions)
#   - Lambda function (Python 3.13 or Node.js 22.x runtime)
#   - CloudWatch log group (created automatically by Lambda on invocation)

set -eE -o pipefail

################################################################################
# Setup
################################################################################

UNIQUE_ID=$(head -c 8 /dev/urandom | od -An -tx1 | tr -d ' ')
FUNCTION_NAME="my-lambda-function-${UNIQUE_ID}"
ROLE_NAME="lambda-execution-role-${UNIQUE_ID}"
LOG_GROUP_NAME="/aws/lambda/${FUNCTION_NAME}"

TEMP_DIR=$(mktemp -d)
```

```
readonly TEMP_DIR
LOG_FILE="${TEMP_DIR}/lambda-gettingstarted.log"

exec > >(tee -a "$LOG_FILE") 2>&1

declare -a CREATED_RESOURCES

################################################################################
# Helper functions
################################################################################

cleanup_resources() {
    # Disable error trap to prevent recursion during cleanup
    trap - ERR
    set +eE

    echo ""
    echo "Cleaning up resources..."
    echo ""

    for ((i=${#CREATED_RESOURCES[@]}-1; i>=0; i--)); do
        local RESOURCE="${CREATED_RESOURCES[$i]}"
        local TYPE="${RESOURCE%%:*}"
        local NAME="${RESOURCE#*:}"

        case "$TYPE" in
            log-group)
                echo "Deleting CloudWatch log group: ${NAME}"
                aws logs delete-log-group \
                    --log-group-name "$NAME" 2>&1 || echo "  WARNING: Could not
 delete log group ${NAME}."
                ;;
            lambda-function)
                echo "Deleting Lambda function: ${NAME}"
                aws lambda delete-function \
                    --function-name "$NAME" 2>&1 || echo "  WARNING: Could not
 delete Lambda function ${NAME}."
                echo "  Waiting for function deletion to complete..."
                local DELETE_WAIT=0
                while aws lambda get-function --function-name "$NAME" > /dev/null
 2>&1; do
                    sleep 2
                    DELETE_WAIT=$((DELETE_WAIT + 2))
                    if [ "$DELETE_WAIT" -ge 60 ]; then
```

```
                        echo "  WARNING: Timed out waiting for function
 deletion."
                        break
                    fi
                done
                ;;
            iam-role-policy)
                local ROLE_PART="${NAME%%|*}"
                local POLICY_PART="${NAME#*|}"
                echo "Detaching policy from role: ${ROLE_PART}"
                aws iam detach-role-policy \
                    --role-name "$ROLE_PART" \
                    --policy-arn "$POLICY_PART" 2>&1 || echo "  WARNING: Could
 not detach policy from role ${ROLE_PART}."
                ;;
            iam-role)
                echo "Deleting IAM role: ${NAME}"
                aws iam delete-role \
                    --role-name "$NAME" 2>&1 || echo "  WARNING: Could not delete
 IAM role ${NAME}."
                ;;
        esac
    done

    if [ -d "$TEMP_DIR" ]; then
        rm -rf "$TEMP_DIR"
    fi

    echo ""
    echo "Cleanup complete."
}

handle_error() {
    echo ""
    echo "========================================="
    echo "ERROR: Script failed at $1"
    echo "========================================="
    echo ""
    if [ ${#CREATED_RESOURCES[@]} -gt 0 ]; then
        echo "Attempting to clean up ${#CREATED_RESOURCES[@]} resource(s)..."
        cleanup_resources
    fi
    exit 1
}
```

```
trap 'handle_error "line $LINENO"' ERR

wait_for_resource() {
    local DESCRIPTION="$1"
    local COMMAND="$2"
    local TARGET_VALUE="$3"
    local TIMEOUT=300
    local ELAPSED=0
    local INTERVAL=5

    echo "Waiting for ${DESCRIPTION}..."
    while true; do
        local RESULT
        RESULT=$(eval "$COMMAND" 2>&1) || true
        if echo "$RESULT" | grep -q "$TARGET_VALUE"; then
            echo "  ${DESCRIPTION} is ready."
            return 0
        fi
        if [ "$ELAPSED" -ge "$TIMEOUT" ]; then
            echo "ERROR: Timed out waiting for ${DESCRIPTION} after ${TIMEOUT}
 seconds."
            return 1
        fi
        sleep "$INTERVAL"
        ELAPSED=$((ELAPSED + INTERVAL))
    done
}

validate_input() {
    local input="$1"
    local pattern="$2"
    if ! [[ "$input" =~ $pattern ]]; then
        echo "ERROR: Invalid input: $input"
        return 1
    fi
    return 0
}

################################################################################
# Region pre-check
################################################################################

CONFIGURED_REGION=$(aws configure get region 2>/dev/null || true)
```

```
if [ -z "$CONFIGURED_REGION" ] && [ -z "$AWS_DEFAULT_REGION" ] && [ -z
 "$AWS_REGION" ]; then
    echo "ERROR: No AWS region configured."
    echo "Run 'aws configure set region <region>' or export AWS_DEFAULT_REGION."
    exit 1
fi

################################################################################
# Runtime selection
################################################################################

echo ""
echo "=========================================="
echo "AWS Lambda - Create Your First Function"
echo "=========================================="
echo ""
echo "Select a runtime for your Lambda function:"
echo "  1) Python 3.13"
echo "  2) Node.js 22.x"
echo ""
echo "Using default: Python 3.13"
RUNTIME_CHOICE="1"

case "$RUNTIME_CHOICE" in
    1)
        RUNTIME="python3.13"
        HANDLER="lambda_function.lambda_handler"
        CODE_FILE="lambda_function.py"
        cat > "${TEMP_DIR}/${CODE_FILE}" << 'PYTHON_EOF'
import json
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def lambda_handler(event, context):
    if not isinstance(event, dict) or 'length' not in event or 'width' not in
 event:
        raise ValueError('Event must contain length and width')
    try:
        length = float(event['length'])
        width = float(event['width'])
        if length < 0 or width < 0:
            raise ValueError('Length and width must be non-negative')
        area = calculate_area(length, width)
        print(f'The area is {area}')
```

```
        logger.info(f'CloudWatch logs group: {context.log_group_name}')
        return json.dumps({'area': area})
    except (TypeError, ValueError) as e:
        logger.error(f'Error processing input: {str(e)}')
        raise
def calculate_area(length, width):
    return length * width
PYTHON_EOF
        echo "Selected runtime: Python 3.13"
        ;;
    2)
        RUNTIME="nodejs22.x"
        HANDLER="index.handler"
        CODE_FILE="index.mjs"
        cat > "${TEMP_DIR}/${CODE_FILE}" << 'NODEJS_EOF'
export const handler = async (event, context) => {
  if (!event || typeof event.length !== 'number' || typeof event.width !==
 'number') {
    throw new Error('Event must contain numeric length and width');
  }
  if (event.length < 0 || event.width < 0) {
    throw new Error('Length and width must be non-negative');
  }
  const area = event.length * event.width;
  console.log(`The area is ${area}`);
  console.log('CloudWatch log group: ', context.logGroupName);
  return JSON.stringify({area});
};
NODEJS_EOF
        echo "Selected runtime: Node.js 22.x"
        ;;
    *)
        echo "ERROR: Invalid choice. Please enter 1 or 2."
        exit 1
        ;;
esac

# ============================================================================
# Step 1: Create IAM execution role
# ============================================================================

echo ""
echo "=========================================="
echo "Step 1: Create IAM execution role"
```

```
echo "============================================="
echo ""

TRUST_POLICY='{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}'

echo "Creating IAM role: ${ROLE_NAME}"
ROLE_OUTPUT=$(aws iam create-role \
    --role-name "$ROLE_NAME" \
    --assume-role-policy-document "$TRUST_POLICY" \
    --query 'Role.Arn' \
    --output text 2>&1)

if ! validate_input "$ROLE_OUTPUT" "^arn:aws:iam::[0-9]+:role/"; then
    echo "ERROR: Failed to create IAM role"
    exit 1
fi

echo "$ROLE_OUTPUT"
ROLE_ARN="$ROLE_OUTPUT"
CREATED_RESOURCES+=("iam-role:${ROLE_NAME}")
echo "Role ARN: ${ROLE_ARN}"

aws iam tag-role \
    --role-name "$ROLE_NAME" \
    --tags Key=project,Value=doc-smith Key=tutorial,Value=lambda-gettingstarted

echo ""
echo "Attaching AWSLambdaBasicExecutionRole policy..."
aws iam attach-role-policy \
    --role-name "$ROLE_NAME" \
    --policy-arn "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole" 2>&1
```

```
CREATED_RESOURCES+=("iam-role-policy:${ROLE_NAME}|arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole")
echo "Policy attached."

# IAM roles can take a few seconds to propagate
echo "Waiting for IAM role to propagate..."
sleep 10

################################################################################
# Step 2: Create Lambda function
################################################################################

echo ""
echo "=========================================="
echo "Step 2: Create Lambda function"
echo "=========================================="
echo ""

echo "Creating deployment package..."
ORIGINAL_DIR=$(pwd)
cd "$TEMP_DIR" || exit 1
zip -j function.zip "$CODE_FILE" > /dev/null 2>&1 || {
    echo "ERROR: Failed to create deployment package"
    exit 1
}
cd "$ORIGINAL_DIR" || exit 1

if [ ! -f "${TEMP_DIR}/function.zip" ]; then
    echo "ERROR: Deployment package creation failed"
    exit 1
fi

echo "Creating Lambda function: ${FUNCTION_NAME}"
echo "  Runtime: ${RUNTIME}"
echo "  Handler: ${HANDLER}"
echo ""

CREATE_OUTPUT=$(aws lambda create-function \
    --function-name "$FUNCTION_NAME" \
    --runtime "$RUNTIME" \
    --role "$ROLE_ARN" \
    --handler "$HANDLER" \
    --architectures x86_64 \
    --zip-file "fileb://${TEMP_DIR}/function.zip" \
```

```
    --tags project=doc-smith,tutorial=lambda-gettingstarted \
    --query '[FunctionName, FunctionArn, Runtime, State]' \
    --output text 2>&1)

if [ -z "$CREATE_OUTPUT" ]; then
    echo "ERROR: Failed to create Lambda function"
    exit 1
fi

echo "$CREATE_OUTPUT"
CREATED_RESOURCES+=("lambda-function:${FUNCTION_NAME}")

wait_for_resource "Lambda function to become Active" \
    "aws lambda get-function-configuration --function-name ${FUNCTION_NAME} --
query State --output text" \
    "Active"

################################################################################
# Step 3: Invoke the function
################################################################################

echo ""
echo "========================================="
echo "Step 3: Invoke the function"
echo "========================================="
echo ""

TEST_EVENT='{"length": 6, "width": 7}'
echo "Invoking function with test event: ${TEST_EVENT}"
echo ""

echo "$TEST_EVENT" > "${TEMP_DIR}/test-event.json"

if ! validate_input "$TEST_EVENT" '"length": [0-9]+, "width": [0-9]+'; then
    echo "ERROR: Invalid test event format"
    exit 1
fi

INVOKE_OUTPUT=$(aws lambda invoke \
    --function-name "$FUNCTION_NAME" \
    --payload "fileb://${TEMP_DIR}/test-event.json" \
    --cli-read-timeout 30 \
    "${TEMP_DIR}/response.json" 2>&1)
echo "$INVOKE_OUTPUT"
```

```
if [ ! -f "${TEMP_DIR}/response.json" ]; then
    echo "ERROR: No response file generated"
    exit 1
fi

RESPONSE=$(cat "${TEMP_DIR}/response.json")
echo ""
echo "Function response: ${RESPONSE}"
echo ""

if echo "$INVOKE_OUTPUT" | grep -qi "functionerror"; then
    echo "WARNING: Function returned an error."
fi

################################################################################
# Step 4: View CloudWatch logs
################################################################################

echo ""
echo "========================================="
echo "Step 4: View CloudWatch Logs"
echo "========================================="
echo ""

echo "Log group: ${LOG_GROUP_NAME}"
echo ""

echo "Waiting for CloudWatch logs to be available..."

LOG_STREAMS=""
for i in $(seq 1 6); do
    LOG_STREAMS=$(aws logs describe-log-streams \
        --log-group-name "$LOG_GROUP_NAME" \
        --order-by LastEventTime \
        --descending \
        --query 'logStreams[0].logStreamName' \
        --output text 2>/dev/null) || true
    if [ -n "$LOG_STREAMS" ] && [ "$LOG_STREAMS" != "None" ]; then
        break
    fi
    LOG_STREAMS=""
    sleep 5
done
```

```
if [ -n "$LOG_STREAMS" ] && [ "$LOG_STREAMS" != "None" ]; then
    echo "Latest log stream: ${LOG_STREAMS}"
    echo ""
    echo "--- Log events ---"
    LOG_EVENTS=$(aws logs get-log-events \
        --log-group-name "$LOG_GROUP_NAME" \
        --log-stream-name "$LOG_STREAMS" \
        --query 'events[].message' \
        --output text 2>&1) || true
    echo "$LOG_EVENTS"
    echo "--- End of log events ---"
else
    echo "No log streams found yet. Logs may take a moment to appear."
    echo "You can view them in the CloudWatch console:"
    echo "  Log group: ${LOG_GROUP_NAME}"
fi

CREATED_RESOURCES+=("log-group:${LOG_GROUP_NAME}")

aws logs tag-log-group \
    --log-group-name "$LOG_GROUP_NAME" \
    --tags project=doc-smith,tutorial=lambda-gettingstarted

################################################################################
# Summary and cleanup
################################################################################

echo ""
echo "========================================="
echo "SUMMARY"
echo "========================================="
echo ""
echo "Resources created:"
echo "  IAM role:          ${ROLE_NAME}"
echo "  Lambda function:   ${FUNCTION_NAME}"
echo "  CloudWatch logs:   ${LOG_GROUP_NAME}"
echo ""
echo "========================================="
echo "CLEANUP"
echo "========================================="
echo ""
echo "Cleaning up all created resources..."
cleanup_resources
```

```
echo ""
echo "Done."
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS CLI - Befehlsreferenz.
  - AttachRolePolicy
  - CreateFunction
  - CreateRole
  - DeleteFunction
  - DeleteLogGroup
  - DeleteRole
  - DescribeLogStreams
  - DetachRolePolicy
  - GetFunction
  - GetFunctionConfiguration
  - GetLogEvents
  - Invoke

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion von einem Browser aus

Das folgende Codebeispiel zeigt, wie ein aufgerufen wird AWS Lambda Funktion von einem Browser aus.

JavaScript

SDK für JavaScript (v2)

Sie können eine browserbasierte Anwendung erstellen, die eine AWS Lambda Funktion zum Aktualisieren einer Amazon DynamoDB-Tabelle mit Benutzerauswahlen.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB
- Lambda

SDK für JavaScript (v3)

Sie können eine browserbasierte Anwendung erstellen, die eine AWS Lambda Funktion zum Aktualisieren einer Amazon DynamoDB-Tabelle mit Benutzerauswahlen. Diese App verwendet AWS SDK für JavaScript v3.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB
- Lambda

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Transformieren von Daten für Ihre Anwendung mit S3 Object Lambda

Das folgende Codebeispiel zeigt, wie Sie Daten für Ihre Anwendung mit S3 Object Lambda transformieren können.

.NET

SDK für .NET

Zeigt, wie Sie benutzerdefinierten Code zu standardmäßigen S3-GET-Anforderungen hinzufügen, um das angeforderte Objekt, das von S3 abgerufen wurde, so zu ändern, dass das Objekt den Anforderungen des anfordernden Clients oder der anfordernden Anwendung entspricht.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- Lambda

- Amazon S3

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion

Die folgenden Codebeispiele zeigen, wie Sie ein erstellen AWS Lambda Funktion, die von Amazon API Gateway aufgerufen wird.

Java

SDK für Java 2.x

Zeigt, wie man eine erstellt AWS Lambda funktioniert mithilfe der Lambda-Java-Laufzeit-API. Dieses Beispiel ruft verschiedene auf AWS Dienste zur Ausführung eines bestimmten Anwendungsfalls. Dieses Beispiel zeigt, wie man eine Lambda-Funktion erstellt, die von Amazon API Gateway aufgerufen wird und eine Amazon-DynamoDB-Tabelle nach Arbeitsjubiläen durchsucht und Amazon Simple Notification Service (Amazon SNS) verwendet, um eine Textnachricht an Ihre Mitarbeiter zu senden, die ihnen zu ihrem einjährigen Jubiläum gratuliert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK für JavaScript (v3)

Zeigt, wie man eine erstellt AWS Lambda Funktion mit dem Lambda JavaScript Laufzeit-API. Dieses Beispiel ruft verschiedene auf AWS Dienste zur Ausführung eines bestimmten

Anwendungsfalls. Dieses Beispiel zeigt, wie man eine Lambda-Funktion erstellt, die von Amazon API Gateway aufgerufen wird und eine Amazon-DynamoDB-Tabelle nach Arbeitsjubiläen durchsucht und Amazon Simple Notification Service (Amazon SNS) verwendet, um eine Textnachricht an Ihre Mitarbeiter zu senden, die ihnen zu ihrem einjährigen Jubiläum gratuliert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Dieses Beispiel ist auch verfügbar im AWS SDK für JavaScript Entwicklerhandbuch für v3.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK für Python (Boto3)

Dieses Beispiel veranschaulicht, wie eine REST-API für Amazon API Gateway erstellt und verwendet wird, die auf eine AWS Lambda -Funktion verweist. Der Lambda-Handler veranschaulicht, wie basierend auf HTTP-Methoden weitergeleitet wird, wie Daten aus der Abfragezeichenfolge, dem Header und dem Text abgerufen werden und wie eine JSON-Antwort zurückgegeben wird.

- Stellen Sie eine Lambda-Funktion bereit.
- REST-API für API Gateway erstellen
- Erstellen Sie eine REST-Ressource, die auf die Lambda-Funktion verweist.
- Erteilen Sie API Gateway die Berechtigung, die Lambda-Funktion aufzurufen.
- Verwenden Sie das Anforderungspaket, um Anforderungen an die REST-API zu senden.
- Bereinigen Sie alle Ressourcen, die während der Demo erstellt wurden.

Dieses Beispiel lässt sich am besten ansehen auf GitHub. Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwenden von Step Functions, um Lambda-Funktionen aufzurufen

Das folgende Codebeispiel zeigt, wie Sie ein erstellen AWS Step Functions Zustandsmaschine, die aufruft AWS Lambda funktioniert nacheinander.

Java

SDK für Java 2.x

Zeigt, wie man eine erstellt AWS serverloser Workflow mithilfe von AWS Step Functions und der AWS SDK for Java 2.x. Jeder Workflow-Schritt wird mit einem implementiert AWS Lambda Funktion.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB
- Lambda
- Amazon SES
- Step-Funktionen

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion

Die folgenden Codebeispiele zeigen, wie Sie ein erstellen AWS Lambda Funktion, die von einem Amazon aufgerufen wird EventBridge geplantes Ereignis.

Java

SDK für Java 2.x

Zeigt, wie ein Amazon erstellt wird EventBridge geplantes Ereignis, das einen aufruft AWS Lambda Funktion. Konfigurieren EventBridge um einen Cron-Ausdruck zu verwenden, um zu planen, wann die Lambda-Funktion aufgerufen wird. In diesem Beispiel erstellen Sie eine Lambda-Funktion mithilfe der Lambda-Java-Laufzeit-API. In diesem Beispiel werden verschiedene aufgerufen AWS Dienste zur Ausführung eines bestimmten Anwendungsfalls. Dieses Beispiel zeigt, wie man eine App erstellt, die eine mobile Textnachricht an Ihre Mitarbeiter sendet, um ihnen zum einjährigen Jubiläum zu gratulieren.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK für JavaScript (v3)

Zeigt, wie man ein Amazon erstellt EventBridge geplantes Ereignis, das einen aufruft AWS Lambda Funktion. Konfigurieren EventBridge um einen Cron-Ausdruck zu verwenden, um zu planen, wann die Lambda-Funktion aufgerufen wird. In diesem Beispiel erstellen Sie eine Lambda-Funktion mithilfe der Lambda-Funktion JavaScript Laufzeit-API. Dieses Beispiel ruft verschiedene auf AWS Dienste zur Ausführung eines bestimmten Anwendungsfalls. Dieses Beispiel zeigt, wie man eine App erstellt, die eine mobile Textnachricht an Ihre Mitarbeiter sendet, um ihnen zum einjährigen Jubiläum zu gratulieren.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Dieses Beispiel ist auch verfügbar im AWS SDK für JavaScript Entwicklerhandbuch für v3.

In diesem Beispiel verwendete Dienste

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK für Python (Boto3)

Dieses Beispiel zeigt, wie man eine registriert AWS Lambda Funktion als Ziel eines geplanten Amazon EventBridge Ereignis. Der Lambda-Handler schreibt eine freundliche Nachricht und die vollständigen Eventdaten an Amazon CloudWatch Protokolle für den späteren Abruf.

- Stellt eine Lambda-Funktion bereit.
- Erzeugt eine EventBridge geplantes Ereignis und macht die Lambda-Funktion zum Ziel.
- Erteilt die Erlaubnis zur Vermietung EventBridge ruft die Lambda-Funktion auf.
- Druckt die neuesten Daten von CloudWatch Protokolliert, um das Ergebnis der geplanten Aufrufe anzuzeigen.
- Bereinigt alle Ressourcen, die während der Demo erstellt wurden.

Dieses Beispiel lässt sich am besten ansehen unter GitHub. Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda

- Amazon SNS

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwenden der Amazon Neptune-API zur Entwicklung einer Lambda-Funktion, die Graphdaten abfragt

Das folgende Codebeispiel zeigt, wie Sie mit der Neptune-API Grafikdaten abfragen.

Java

SDK für Java 2.x

Zeigt, wie die Java-API von Amazon Neptune verwendet wird, um eine Lambda-Funktion zu erstellen, die Grafikdaten innerhalb der VPC abfragt.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- Lambda
- Neptune

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von Eigenschaftsvariablen in Überwachungs-Dashboards zur Überwachung mehrerer serverloser Funktionen

Wie das aussehen kann, sehen Sie am nachfolgenden Beispielcode:

- Erstellen Sie Lambda-Funktionen für die Überwachung
- Erstellen Sie eine CloudWatch Dashboard
- Fügen Sie dem Dashboard eine Eigenschaftsvariable hinzu
- Bereinigen von Ressourcen

Bash

AWS CLI mit Bash-Skript

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie es im Tutorials und Beispiele-Repository für Entwickler einrichten und ausführen.

```
#!/bin/bash

# CloudWatch Dashboard with Lambda Function Variable Script
# This script creates a CloudWatch dashboard with a property variable for Lambda
 function names

set -euo pipefail

# Security: Set restrictive umask
umask 0077

# Set up logging with secure permissions
LOG_FILE="cloudwatch-dashboard-script-v4.log"
touch "$LOG_FILE"
chmod 600 "$LOG_FILE"
echo "Starting script execution at $(date)" >> "$LOG_FILE"

# Function to log commands and their output (with sensitive data sanitization)
log_cmd() {
    local cmd="$1"
    local sanitized_cmd="${cmd//--password*/--password [REDACTED]}"
    sanitized_cmd="${sanitized_cmd//--secret*/--secret [REDACTED]}"
    echo "$(date): Running command: $sanitized_cmd" >> "$LOG_FILE"
    eval "$cmd" 2>&1 | tee -a "$LOG_FILE"
    return ${PIPESTATUS[0]}
}

# Function to check for errors in command output
check_error() {
    local cmd_output="$1"
    local cmd_status="$2"
    local error_msg="$3"
```

```
    if [ $cmd_status -ne 0 ] || echo "$cmd_output" | grep -qi "error"; then
        echo "ERROR: $error_msg" | tee -a "$LOG_FILE"
        # Sanitize output before logging
        local sanitized_output="${cmd_output//arn:aws:iam::[0-9]*/
arn:aws:iam::ACCOUNT_ID}"
        echo "Command output: $sanitized_output" | tee -a "$LOG_FILE"
        cleanup_resources
        exit 1
    fi
}

# Trap errors and cleanup
trap 'cleanup_resources' EXIT ERR INT TERM

# Function to clean up resources
cleanup_resources() {
    local exit_code=$?

    echo "" | tee -a "$LOG_FILE"
    echo "========================================" | tee -a "$LOG_FILE"
    echo "CLEANUP PROCESS" | tee -a "$LOG_FILE"
    echo "========================================" | tee -a "$LOG_FILE"

    if [ -n "${DASHBOARD_NAME:-}" ]; then
        echo "Deleting CloudWatch dashboard: $DASHBOARD_NAME" | tee -a
 "$LOG_FILE"
        aws cloudwatch delete-dashboards --dashboard-names "$DASHBOARD_NAME" 2>&1
 >> "$LOG_FILE" || true
    fi

    if [ -n "${LAMBDA_FUNCTION1:-}" ]; then
        echo "Deleting Lambda function: $LAMBDA_FUNCTION1" | tee -a "$LOG_FILE"
        aws lambda delete-function --function-name "$LAMBDA_FUNCTION1" 2>&1 >>
 "$LOG_FILE" || true
    fi

    if [ -n "${LAMBDA_FUNCTION2:-}" ]; then
        echo "Deleting Lambda function: $LAMBDA_FUNCTION2" | tee -a "$LOG_FILE"
        aws lambda delete-function --function-name "$LAMBDA_FUNCTION2" 2>&1 >>
 "$LOG_FILE" || true
    fi

    if [ -n "${ROLE_NAME:-}" ]; then
```

```
        echo "Detaching policy from role: $ROLE_NAME" | tee -a "$LOG_FILE"
        aws iam detach-role-policy --role-name "$ROLE_NAME" --policy-arn
 arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole 2>&1 >>
 "$LOG_FILE" || true

        echo "Deleting IAM role: $ROLE_NAME" | tee -a "$LOG_FILE"
        aws iam delete-role --role-name "$ROLE_NAME" 2>&1 >> "$LOG_FILE" || true
    fi

    # Clean up temporary files securely
    shred -vfz -n 3 trust-policy.json lambda_function.py lambda_function.zip 2>/
dev/null || rm -f trust-policy.json lambda_function.py lambda_function.zip

    echo "Cleanup completed." | tee -a "$LOG_FILE"

    return $exit_code
}

# Validate AWS CLI is installed and authenticated
if ! command -v aws &> /dev/null; then
    echo "ERROR: AWS CLI is not installed" | tee -a "$LOG_FILE"
    exit 1
fi

if ! aws sts get-caller-identity &> /dev/null; then
    echo "ERROR: AWS CLI is not properly authenticated" | tee -a "$LOG_FILE"
    exit 1
fi

# Get AWS region with validation
AWS_REGION=$(aws configure get region 2>/dev/null || echo "")
if [ -z "$AWS_REGION" ]; then
    AWS_REGION="us-east-1"
    echo "No region found in AWS config, defaulting to $AWS_REGION" | tee -a
 "$LOG_FILE"
else
    echo "Using AWS region: $AWS_REGION" | tee -a "$LOG_FILE"
fi

# Validate region format
if ! [[ "$AWS_REGION" =~ ^[a-z]{2}-[a-z]+-[0-9]$ ]]; then
    echo "ERROR: Invalid AWS region format: $AWS_REGION" | tee -a "$LOG_FILE"
    exit 1
fi
```

```
# Generate unique identifiers using secure random with validation
RANDOM_ID=$(openssl rand -hex 6)
if [ -z "$RANDOM_ID" ] || [ ${#RANDOM_ID} -ne 12 ]; then
    echo "ERROR: Failed to generate valid random identifier" | tee -a "$LOG_FILE"
    exit 1
fi

DASHBOARD_NAME="LambdaMetricsDashboard-${RANDOM_ID}"
LAMBDA_FUNCTION1="TestFunction1-${RANDOM_ID}"
LAMBDA_FUNCTION2="TestFunction2-${RANDOM_ID}"
ROLE_NAME="LambdaExecutionRole-${RANDOM_ID}"

# Validate resource names don't exceed AWS limits
if [ ${#DASHBOARD_NAME} -gt 128 ] || [ ${#LAMBDA_FUNCTION1} -gt 64 ] ||
 [ ${#ROLE_NAME} -gt 64 ]; then
    echo "ERROR: Generated resource names exceed AWS limits" | tee -a "$LOG_FILE"
    exit 1
fi

echo "Using random identifier: $RANDOM_ID" | tee -a "$LOG_FILE"
echo "Dashboard name: $DASHBOARD_NAME" | tee -a "$LOG_FILE"
echo "Lambda function names: $LAMBDA_FUNCTION1, $LAMBDA_FUNCTION2" | tee -a
 "$LOG_FILE"
echo "IAM role name: $ROLE_NAME" | tee -a "$LOG_FILE"

# Create IAM role for Lambda functions
echo "Creating IAM role for Lambda..." | tee -a "$LOG_FILE"
TRUST_POLICY='{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}'

echo "$TRUST_POLICY" > trust-policy.json
chmod 600 trust-policy.json
```

```
# Validate JSON before use
if ! python3 -m json.tool trust-policy.json > /dev/null 2>&1; then
    echo "ERROR: Invalid trust policy JSON" | tee -a "$LOG_FILE"
    exit 1
fi

ROLE_OUTPUT=$(log_cmd "aws iam create-role --role-name '$ROLE_NAME' --assume-
role-policy-document file://trust-policy.json --tags Key=project,Value=doc-smith
 Key=tutorial,Value=cloudwatch-streams --output json")
check_error "$ROLE_OUTPUT" $? "Failed to create IAM role"

ROLE_ARN=$(echo "$ROLE_OUTPUT" | python3 -c "import sys, json;
 print(json.load(sys.stdin)['Role']['Arn'])" 2>/dev/null)
if [ -z "$ROLE_ARN" ]; then
    echo "ERROR: Failed to extract Role ARN" | tee -a "$LOG_FILE"
    exit 1
fi
echo "Role ARN: $ROLE_ARN" | tee -a "$LOG_FILE"

# Attach Lambda basic execution policy to the role
echo "Attaching Lambda execution policy to role..." | tee -a "$LOG_FILE"
POLICY_OUTPUT=$(log_cmd "aws iam attach-role-policy --role-name '$ROLE_NAME' --
policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole")
check_error "$POLICY_OUTPUT" $? "Failed to attach policy to role"

# Wait for role to propagate
echo "Waiting for IAM role to propagate..." | tee -a "$LOG_FILE"
sleep 10

# Create simple Python Lambda function code with security validation
echo "Creating Lambda function code..." | tee -a "$LOG_FILE"
cat > lambda_function.py << 'LAMBDA_EOF'
def handler(event, context):
    print("Lambda function executed successfully")
    return {
        'statusCode': 200,
        'body': 'Success'
    }
LAMBDA_EOF

chmod 600 lambda_function.py

# Validate Python syntax
if ! python3 -m py_compile lambda_function.py 2>/dev/null; then
```

```
    echo "ERROR: Invalid Python syntax in Lambda function" | tee -a "$LOG_FILE"
    exit 1
fi

# Zip the Lambda function code
zip -j -q lambda_function.zip lambda_function.py
if [ ! -f lambda_function.zip ]; then
    echo "ERROR: Failed to create lambda_function.zip" | tee -a "$LOG_FILE"
    exit 1
fi
chmod 600 lambda_function.zip

# Validate zip file integrity
if ! unzip -t lambda_function.zip > /dev/null 2>&1; then
    echo "ERROR: Created zip file is corrupted" | tee -a "$LOG_FILE"
    exit 1
fi

# Create first Lambda function
echo "Creating first Lambda function: $LAMBDA_FUNCTION1..." | tee -a "$LOG_FILE"
LAMBDA1_OUTPUT=$(log_cmd "aws lambda create-function --function-name
 '$LAMBDA_FUNCTION1' --runtime python3.11 --role '$ROLE_ARN' --handler
 lambda_function.handler --zip-file fileb://lambda_function.zip --timeout 30 --
memory-size 128 --tags project=doc-smith,tutorial=cloudwatch-streams")
check_error "$LAMBDA1_OUTPUT" $? "Failed to create first Lambda function"

# Create second Lambda function
echo "Creating second Lambda function: $LAMBDA_FUNCTION2..." | tee -a "$LOG_FILE"
LAMBDA2_OUTPUT=$(log_cmd "aws lambda create-function --function-name
 '$LAMBDA_FUNCTION2' --runtime python3.11 --role '$ROLE_ARN' --handler
 lambda_function.handler --zip-file fileb://lambda_function.zip --timeout 30 --
memory-size 128 --tags project=doc-smith,tutorial=cloudwatch-streams")
check_error "$LAMBDA2_OUTPUT" $? "Failed to create second Lambda function"

# Invoke Lambda functions to generate some metrics
echo "Invoking Lambda functions to generate metrics..." | tee -a "$LOG_FILE"
log_cmd "aws lambda invoke --function-name '$LAMBDA_FUNCTION1' --payload '{}' /
dev/null" || true
log_cmd "aws lambda invoke --function-name '$LAMBDA_FUNCTION2' --payload '{}' /
dev/null" || true

# Create CloudWatch dashboard with property variable
echo "Creating CloudWatch dashboard with property variable..." | tee -a
 "$LOG_FILE"
```

```
# Create dashboard body with proper escaping and validation
DASHBOARD_BODY=$(cat <<EOF
{
  "widgets": [
    {
      "type": "metric",
      "x": 0,
      "y": 0,
      "width": 12,
      "height": 6,
      "properties": {
        "metrics": [
          [ "AWS/Lambda", "Invocations", "FunctionName", "$LAMBDA_FUNCTION1" ]
        ],
        "view": "timeSeries",
        "stacked": false,
        "region": "$AWS_REGION",
        "title": "Lambda Invocations",
        "period": 300,
        "stat": "Sum"
      }
    }
  ]
}
EOF
)

# Validate JSON before sending
if ! echo "$DASHBOARD_BODY" | python3 -m json.tool > /dev/null 2>&1; then
    echo "ERROR: Dashboard body is not valid JSON" | tee -a "$LOG_FILE"
    exit 1
fi

# First create a basic dashboard without variables
echo "Creating initial dashboard without variables..." | tee -a "$LOG_FILE"
DASHBOARD_OUTPUT=$(aws cloudwatch put-dashboard --dashboard-name
 "$DASHBOARD_NAME" --dashboard-body "$DASHBOARD_BODY" --output json 2>&1)
check_error "$DASHBOARD_OUTPUT" $? "Failed to create initial CloudWatch
 dashboard"

# Now let's try to add a property variable using the console instructions
echo "To complete the tutorial, please follow these steps in the CloudWatch
 console:" | tee -a "$LOG_FILE"
```

```
echo "1. Open the CloudWatch console at https://console.aws.amazon.com/
cloudwatch/" | tee -a "$LOG_FILE"
echo "2. Navigate to Dashboards and select your dashboard: $DASHBOARD_NAME" | tee
 -a "$LOG_FILE"
echo "3. Choose Actions > Variables > Create a variable" | tee -a "$LOG_FILE"
echo "4. Choose Property variable" | tee -a "$LOG_FILE"
echo "5. For Property that the variable changes, choose FunctionName" | tee -a
 "$LOG_FILE"
echo "6. For Input type, choose Select menu (dropdown)" | tee -a "$LOG_FILE"
echo "7. Choose Use the results of a metric search" | tee -a "$LOG_FILE"
echo "8. Choose Pre-built queries > Lambda > Errors" | tee -a "$LOG_FILE"
echo "9. Choose By Function Name and then choose Search" | tee -a "$LOG_FILE"
echo "10. (Optional) Configure any secondary settings as desired" | tee -a
 "$LOG_FILE"
echo "11. Choose Add variable" | tee -a "$LOG_FILE"
echo "" | tee -a "$LOG_FILE"
echo "The dashboard has been created and can be accessed at:" | tee -a
 "$LOG_FILE"
echo "https://console.aws.amazon.com/cloudwatch/home#dashboards:name=
$DASHBOARD_NAME" | tee -a "$LOG_FILE"

# Verify dashboard creation
echo "Verifying dashboard creation..." | tee -a "$LOG_FILE"
VERIFY_OUTPUT=$(aws cloudwatch get-dashboard --dashboard-name "$DASHBOARD_NAME"
 --output json 2>&1)
check_error "$VERIFY_OUTPUT" $? "Failed to verify dashboard creation"

echo "" | tee -a "$LOG_FILE"
echo "==========================================" | tee -a "$LOG_FILE"
echo "DASHBOARD CREATED SUCCESSFULLY" | tee -a "$LOG_FILE"
echo "==========================================" | tee -a "$LOG_FILE"
echo "Dashboard Name: $DASHBOARD_NAME" | tee -a "$LOG_FILE"
echo "Lambda Functions: $LAMBDA_FUNCTION1, $LAMBDA_FUNCTION2" | tee -a
 "$LOG_FILE"
echo "" | tee -a "$LOG_FILE"
echo "You can view your dashboard in the CloudWatch console:" | tee -a
 "$LOG_FILE"
echo "https://console.aws.amazon.com/cloudwatch/home#dashboards:name=
$DASHBOARD_NAME" | tee -a "$LOG_FILE"
echo "" | tee -a "$LOG_FILE"

# Auto-confirm cleanup
echo "" | tee -a "$LOG_FILE"
echo "==========================================" | tee -a "$LOG_FILE"
```

```
echo "CLEANUP CONFIRMATION" | tee -a "$LOG_FILE"
echo "========================================" | tee -a "$LOG_FILE"
echo "The following resources were created:" | tee -a "$LOG_FILE"
echo "- CloudWatch Dashboard: $DASHBOARD_NAME" | tee -a "$LOG_FILE"
echo "- Lambda Function: $LAMBDA_FUNCTION1" | tee -a "$LOG_FILE"
echo "- Lambda Function: $LAMBDA_FUNCTION2" | tee -a "$LOG_FILE"
echo "- IAM Role: $ROLE_NAME" | tee -a "$LOG_FILE"
echo "" | tee -a "$LOG_FILE"
echo "Auto-confirming cleanup of all created resources..." | tee -a "$LOG_FILE"

echo "Script completed successfully." | tee -a "$LOG_FILE"
exit 0
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS CLI - Befehlsreferenz.
  - AttachRolePolicy
  - CreateFunction
  - CreateRole
  - DeleteDashboards
  - DeleteFunction
  - DeleteRole
  - DetachRolePolicy
  - GetDashboard
  - Invoke
  - PutDashboard

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Schreiben Sie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito-Benutzerauthentifizierung mithilfe einer AWS SDK

Das folgende Codebeispiel veranschaulicht, wie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon-Cognito-Benutzerauthentifizierung geschrieben werden.

- Verwenden Sie Administratorfunktionen, um einen Benutzer zu einem Benutzerpool hinzuzufügen.
- Konfigurieren Sie einen Benutzerpool, um eine Lambda-Funktion für den `PostAuthentication`-Trigger aufzurufen.
- Melden Sie den neuen Benutzer bei Amazon Cognito an.
- Die Lambda-Funktion schreibt benutzerdefinierte Informationen in CloudWatch Protokolle und in eine DynamoDB-Tabelle.
- Ermitteln Sie benutzerdefinierte Daten aus der DynamoDB-Tabelle, zeigen Sie diese an und bereinigen Sie dann die Ressourcen.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Führen Sie ein interaktives Szenario an einem Prompt aus.

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
```

```
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(ctx context.Context, userPoolId string,
 tableName string) (string, string) {
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(ctx, userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(ctx, userPoolId, user.UserName,
 password)
```

```
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   pwSet = true
  }
 }

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(ctx context.Context, userPoolId
 string, activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
```

```
func (runner *ActivityLog) SignInUser(ctx context.Context, clientId string,
 userName string, password string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
 runner.questioner.Ask("Press Enter when you're ready.")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")

 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(ctx context.Context, tableName
 string, userName string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
```

```
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])
 userName, password := runner.AddUserToPool(ctx, stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(ctx, stackOutputs["TableName"], userName)

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Behandeln Sie den `PostAuthentication`-Trigger mit einer Lambda-Funktion.

```
import (
 "context"
 "fmt"
 "log"
 "os"
 "time"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
```

```
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
   TableName: aws.String(tableName),
  })
  if err != nil {
   log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
  } else {
   log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
  }
 }

 return event, nil
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
```

```
 }
 lambda.Start(h.HandleRequest)
}
```

Erstellen Sie eine Struktur, die allgemeine Aufgaben ausführt.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
```

```
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
```

```
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
```

```
 log.Println(strings.Repeat("-", 88))
}
```

Erstellen Sie eine Struktur, die Amazon Cognito-Aktionen umschließt.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
```

```
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
```

```
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}
```

```
// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}
```

```
// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
```

```
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Erstellen Sie eine Struktur, die DynamoDB-Aktionen umhüllt.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
```

```
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
```

```
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
```

```
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Erstellen Sie eine Struktur, die umschließt CloudWatch Protokolliert Aktionen.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
```

```
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Erstellen Sie eine Struktur, die umschließt CloudFormation Aktionen.

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
```

```
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Bereinigen von Ressourcen.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
```

```
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
```

```
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Serverless-Beispiele für Lambda

Die folgenden Codebeispiele zeigen, wie Lambda verwendet wird mit AWS SDKs.

Beispiele

- Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion
- Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser
- Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser
- Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser
- Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser
- Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser
- Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen
- Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser
- Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser

- Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser
- Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die eine Verbindung zu einer RDS-Datenbank herstellt. Die Funktion stellt eine einfache Datenbankanfrage und gibt das Ergebnis zurück.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion mithilfe von .NET.

```
using System.Data;
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
```

```
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
        string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

 $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


        try
        {
            await using var connection = new MySqlConnection(connectionString);
            await connection.OpenAsync();

            const string sql = "SELECT @param1 + @param2 AS Sum";
```

```
            await using var command = new MySqlCommand(sql, connection);
            command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
 "0"));
            command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
 "0"));

            await using var reader = await command.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                int result = reader.GetInt32("Sum");

                //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
 sum is: 45\"}","isBase64Encoded":false}
                return new APIGatewayProxyResponse
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
 {result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Go.

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
```

```
var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
var region string = os.Getenv("AWS_REGION")

cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
 panic("configuration error: " + err.Error())
}

authenticationToken, err := auth.BuildAuthToken(
 context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
if err != nil {
 panic("failed to create authentication token: " + err.Error())
}

dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
 dbUser, authenticationToken, dbEndpoint, dbName,
)

db, err := sql.Open("mysql", dsn)
if err != nil {
 panic(err)
}

defer db.Close()

var sum int
err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
if err != nil {
 panic(err)
}
s := fmt.Sprint(sum)
message := fmt.Sprintf("The selected sum is: %s", s)

messageBytes, err := json.Marshal(message)
if err != nil {
 return nil, err
}

messageString := string(messageBytes)
return map[string]interface{}{
 "statusCode": 200,
 "headers":    map[string]string{"Content-Type": "application/json"},
 "body":       messageString,
}, nil
```

```
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
```

```
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
```

```
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mit JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
```

```
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}
```

```
export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mit TypeScript.

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}
```

```
async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit PHP.

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
```

```
            'region' => getenv('AWS_REGION'),
        ];

        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
```

```
    }

    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Python.

```
import json
import os
import boto3
import pymysql
```

```
# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Ruby.

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )
```

```
  begin
    conn = Mysql2::Client.new(
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit Rust.

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
```

```
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
```

```
        db_user = db_username
    );

    let signable_request =
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
```

```
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die ein durch den Empfang von Datensätzen aus einem Kinesis-Stream ausgelöstes Ereignis empfängt. Die Funktion ruft die Kinesis-Nutzlast ab, dekodiert von Base64 und protokolliert den Datensatzinhalt.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
```

```
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
```

```
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
```

```
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines Kinesis-Ereignisses mit Lambda mithilfe von JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
```

```
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Ein Kinesis-Ereignis mit Lambda nutzen TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
```

```
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
```

```
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
```

```
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die ein durch den Empfang von Datensätzen aus einem DynamoDB-Stream ausgelöstes Ereignis empfängt. Die Funktion ruft die DynamoDB-Nutzdaten ab und protokolliert den Inhalt des Datensatzes.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein DynamoDB-Ereignis mit Lambda verwenden JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Ein DynamoDB-Ereignis mit Lambda verwenden TypeScript.

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data
```

```
            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Python.

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())
```

```
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die ein durch den Empfang von Datensätzen aus einem DocumentDB-Änderungsstream ausgelöstes Ereignis empfängt. Die Funktion ruft die DocumentDB-Nutzdaten ab und protokolliert den Inhalt des Datensatzes.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von .NET.

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
 ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument, new
 JsonSerializerOptions { WriteIndented = true });
```

```
        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }

        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
```

```
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
```

```
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Go.

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
```

```
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Java.

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
```

```
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");

        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzung eines Amazon DocumentDB-Ereignisses mit Lambda JavaScript.

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Nutzung eines Amazon DocumentDB-Ereignisses mit Lambda TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
```

```
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];
```

```
        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument, JSON_PRETTY_PRINT) .
 "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Python.

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
```

```
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Ruby.

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_document_db_event(record);
```

```
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser

Die folgenden Codebeispiele zeigen, wie eine Lambda-Funktion implementiert wird, die ein Ereignis empfängt, das durch den Empfang von Datensätzen von einem Amazon MSK-Cluster ausgelöst wird. Die Funktion ruft die MSK-Nutzdaten ab und protokolliert den Inhalt des Datensatzes.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von .NET.

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);
```

```
                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Go.

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
```

```
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);
```

```
                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzung eines Amazon MSK-Ereignisses mit Lambda JavaScript.

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

Nutzung eines Amazon MSK-Ereignisses mit Lambda TypeScript.

```
import { MSKEvent, Context } from "aws-lambda";
```

```
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Python.

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Ruby.

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von Rust.

```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
```

```
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
        }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser

In den folgenden Codebeispiele wird die Implementierung einer Lambda-Funktion gezeigt, die ein Ereignis empfängt, das durch Hochladen eines Objekts in einen S3-Bucket ausgelöst wird. Die Funktion ruft den Namen des S3-Buckets sowie den Objektschlüssel aus dem Ereignisparameter ab und ruft die Amazon-S3-API auf, um den Inhaltstyp des Objekts abzurufen und zu protokollieren.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von .NET

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");
```

```
                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Go

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
```

```
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Java

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
```

```
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzung eines S3-Ereignisses mit Lambda JavaScript.

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
```

```
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

Nutzung eines S3-Ereignisses mit Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von PHP.

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
```

```
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Python

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
```

```
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Ruby.

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
```

```
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Rust

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();
```

```
    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
 SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
 name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
 exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
```

```
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen

In den folgenden Codebeispiele wird die Implementierung einer Lambda-Funktion veranschaulicht, die ein Ereignis empfängt, das durch das Empfangen von Nachrichten aus einem SNS-Thema ausgelöst wird. Die Funktion ruft die Nachrichten aus dem Ereignisparameter ab und protokolliert den Inhalt jeder Nachricht.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
```

```
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
{record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzung eines SNS-Ereignisses mit Lambda JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Nutzung eines SNS-Ereignisses mit Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";
```

```
export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von PHP

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
```

```
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von Rust

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
```

```
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser

Die folgenden Codebeispiele zeigen, wie eine Lambda-Funktion implementiert wird, die ein Ereignis empfängt, das durch den Empfang von Nachrichten aus einer SQS-Warteschlange ausgelöst wird. Die Funktion ruft die Nachrichten aus dem Ereignisparameter ab und protokolliert den Inhalt jeder Nachricht.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
```

```
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
```

```
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
```

```
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }

    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzung eines SQS-Ereignisses mit Lambda JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
```

```
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Nutzung eines SQS-Ereignisses mit Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser

Die folgenden Codebeispiele zeigen, wie eine teilweise Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse von einem Kinesis-Stream empfangen. Die Funktion meldet die

Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
```

```
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

```
public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""
```

```
  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
```

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Meldung von Kinesis-Batch-Elementfehlern mit Lambda mithilfe von TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
```

```
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
```

```
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
```

```
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();
```

```
    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser

Die folgenden Codebeispiele zeigen, wie eine teilweise Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse von einem DynamoDB-Stream empfangen. Die Funktion meldet die Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;
```

```
public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
```

```
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
```

```
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Meldung von DynamoDB-Batch-Elementfehlern mit Lambda mithilfe von JavaScript.

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

Meldung von DynamoDB-Batch-Elementfehlern mit Lambda mithilfe von TypeScript.

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }
```

```
  return { batchItemFailures: batchItemFailures };
};
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
```

```
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;
```

```
    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }
```

```
    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser

Die folgenden Codebeispiele zeigen, wie eine teilweise Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse aus einer SQS-Warteschlange empfangen. Die Funktion meldet die Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
```

```
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
```

```
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
```

```
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {
         List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();

         for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
             try {
                 //process your message
             } catch (Exception e) {
                 //Add failed message identifier to the batchItemFailures list
                 batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
             }
         }
         return new SQSBatchResponse(batchItemFailures);
     }
}
```

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Meldung von SQS-Batch-Elementfehlern mit Lambda mithilfe von JavaScript.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
```

```
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

Meldung von SQS-Batch-Elementfehlern mit Lambda mithilfe von TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();
```

```
        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
```

```
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

AWS Beiträge der Community für Lambda

AWS Beiträge der Gemeinschaft sind Beispiele, die von mehreren Teams auf der ganzen Welt erstellt wurden und verwaltet werden AWS. Um Feedback zu geben, verwenden Sie den Mechanismus, der in den verlinkten Repositorys bereitgestellt wird.

Beispiele

- Erstellen und Testen einer Serverless-Anwendung

Erstellen und Testen einer Serverless-Anwendung

Die folgenden Codebeispiele zeigen, wie eine Serverless-Anwendung mithilfe von API Gateway mit Lambda und DynamoDB erstellt und getestet wird.

.NET

SDK für .NET

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des .NET SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB

- Lambda

Go

SDK für Go V2

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des Go SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Java

SDK für Java 2.x

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des Java SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Rust

SDK für Rust

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des Rust SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Für eine vollständige Liste von AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter Verwenden von Lambda mit einem AWS SDK. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

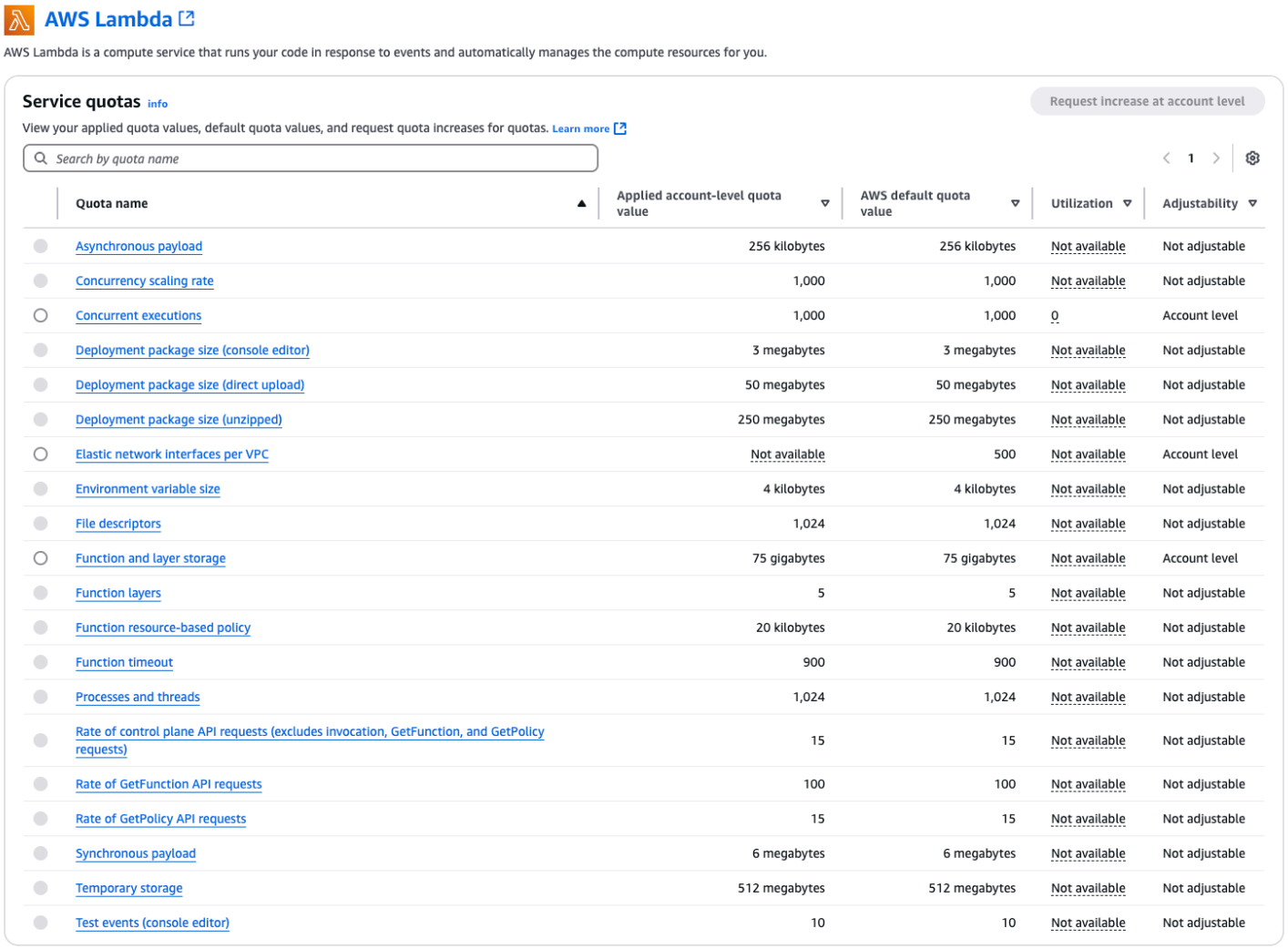
Lambda-Kontingente

> **Important**
>
> Neu AWS-Konten haben die Parallelität und die Speicherkontingente für Lambda-Funktionen und Lambda-MicroVMs reduziert und die Kontingente für vCPU- und Anforderungsraten für Lambda-Webfunktionen reduziert. AWS erhöht diese Kontingente automatisch auf der Grundlage Ihrer Nutzung.

AWS Lambda ist so konzipiert, dass es schnell skaliert wird, um der Nachfrage gerecht zu werden, sodass Ihre Funktionen skaliert werden können, um den Traffic in Ihrer Anwendung zu bedienen. Lambda ist für kurzlebige Rechenaufgaben konzipiert, die den Status zwischen Aufrufen nicht beibehalten oder nicht darauf angewiesen sind. Code kann in einem einzigen Aufruf bis zu 15 Minuten lang ausgeführt werden (bis zu 90 Minuten für Lambda Managed Instances-Funktionen, die asynchron oder über eine beliebige Ereignisquellenzuordnung außer Amazon MQ und Amazon DocumentDB aufgerufen werden), und eine einzelne Funktion kann bis zu 10.240 MB Arbeitsspeicher beanspruchen.

Es ist wichtig, die Schutzmaßnahmen zu verstehen, die zum Schutz Ihres Kontos und der Arbeitsbelastung anderer Kunden eingerichtet wurden. Insgesamt gibt es Servicekontingente AWS Dienstleistungen und bestehen aus harten Grenzwerten, die Sie nicht ändern können, und weichen Grenzwerten, für die Sie Erhöhungen beantragen können. Standardmäßig wird allen neuen Konten ein Kontingentprofil zugewiesen, das die Erkundung folgender Elemente ermöglicht AWS Dienste.

Um die Kontingente zu sehen, die für Ihr Konto gelten, navigieren Sie zum Service-Quotas-Dashboard. Hier können Sie Service Quotas anzeigen, eine Erhöhung des Service-Kontingents anfordern und die aktuelle Auslastung anzeigen. Von hier aus können Sie einen Drilldown zu einem bestimmten Objekt durchführen AWS Dienst, wie Lambda:



AWS Lambda

AWS Lambda is a compute service that runs your code in response to events and automatically manages the compute resources for you.

Service quotas info

Request increase at account level

View your applied quota values, default quota values, and request quota increases for quotas. Learn more

Search by quota name

1

Quota name	Applied account-level quota value	AWS default quota value	Utilization	Adjustability
Asynchronous payload	256 kilobytes	256 kilobytes	Not available	Not adjustable
Concurrency scaling rate	1,000	1,000	Not available	Not adjustable
Concurrent executions	1,000	1,000	0	Account level
Deployment package size (console editor)	3 megabytes	3 megabytes	Not available	Not adjustable
Deployment package size (direct upload)	50 megabytes	50 megabytes	Not available	Not adjustable
Deployment package size (unzipped)	250 megabytes	250 megabytes	Not available	Not adjustable
Elastic network interfaces per VPC	Not available	500	Not available	Account level
Environment variable size	4 kilobytes	4 kilobytes	Not available	Not adjustable
File descriptors	1,024	1,024	Not available	Not adjustable
Function and layer storage	75 gigabytes	75 gigabytes	Not available	Account level
Function layers	5	5	Not available	Not adjustable
Function resource-based policy	20 kilobytes	20 kilobytes	Not available	Not adjustable
Function timeout	900	900	Not available	Not adjustable
Processes and threads	1,024	1,024	Not available	Not adjustable
Rate of control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15	15	Not available	Not adjustable
Rate of GetFunction API requests	100	100	Not available	Not adjustable
Rate of GetPolicy API requests	15	15	Not available	Not adjustable
Synchronous payload	6 megabytes	6 megabytes	Not available	Not adjustable
Temporary storage	512 megabytes	512 megabytes	Not available	Not adjustable
Test events (console editor)	10	10	Not available	Not adjustable

In folgenden Abschnitten sind Standardkontingente und Grenzwerte in Lambda nach Kategorien aufgeführt.

Themen

- Datenverarbeitung und Speicherung
- Funktionskonfiguration, -bereitstellung und -ausführung
- Lambda-API-Anforderungen
- Dauerhafte Funktionen
- Lambda-MicroVMs
- Lambda-Web-Funktionen
- Sonstige Services

Datenverarbeitung und Speicherung

Lambda legt Kontingente für die Menge an Datenverarbeitung und Speicherressourcen, die Sie verwenden können, um Funktionen auszuführen und zu speichern. Kontingente für gleichzeitige Ausführungen und Speicherung gelten pro AWS-Region. Die Elastic-Network-Schnittstelle (ENI)-Kontingente gelten für jede Virtual Private Cloud (VPC), unabhängig von der Region. Die folgenden Kontingente können gegenüber ihren Standardwerten erhöht werden. Weitere Informationen finden Sie unter Beantragen einer Kontingenterhöhung im Service-Quotas-Benutzerhandbuch.

Ressource	Standardkontingent	Kann erhöht werden bis zu
Gleichzeitige Ausführungen	1.000	Zehntausende
Speicher für hochgeladene Funktionen (.zip-Dateiarchive) und Ebenen mit Lambda-managed Speicher. Jede Funktionsversion und Ebenenversion verbraucht Speicher. Um Speicherbeschränkungen zu vermeiden, können Sie Ihre Funktionen und Ebenen so konfigurieren, dass sie verwendet werden selbstverwalteter S3-Codespeicher stattdessen. Bewährte Methoden für die Verwaltung Ihres Codespeichers finden Sie bei Serverless Land unter Monitoring Lambda code storage.	300 GB (entpackt)	Nicht erhöhbar. Benutzen selbstverwalteter S3-Codespeicher für die Speicherung über diesem Limit hinaus.
Speicher für als Container-Images definierten Funktionen Diese Bilder werden in Amazon ECR gespeichert.	Siehe Amazon-ECR-Service kontingente.	
Elastic Network-Schnittstellen in der Virtual Private Cloud (VPC)	3,000	Tausende

Ressource	Standardkontingent	Kann erhöht werden bis zu
Note Dieses Kontingent wird mit anderen Services wie Amazon Elastic File System (Amazon EFS) geteilt. Siehe Amazon-VPC-Kontingente.		

Weitere Details zur Gleichzeitigkeit und zur datenverkehrbasierten Skalierung der Funktionsgleichzeitigkeit von Lambda finden Sie unter Die Skalierung von Lambda-Ereignisfunktionen verstehen.

Funktionskonfiguration, -bereitstellung und -ausführung

Die folgenden Kontingente gelten für die Konfiguration, Bereitstellung und Ausführung von Funktionen. Sofern nicht anders angegeben, können sie nicht geändert werden.

Note

Die Lambda-Dokumentation, die Protokollmeldungen und die Konsole verwenden die Abkürzung MB (anstelle von MiB), um auf 1 024 KB zu verweisen.

Ressource	Quota
Funktion Speicherzuweisung	128 MB bis 10.240 MB (in Schritten von 1 MB). Hinweis: Lambda weist die CPU-Leistung proportional zur Menge des konfigurierten Arbeitsspeichers zu. Sie können den Arbeitsspeicher und die CPU-Leistung, die Ihrer Funktion zugewiesen sind, mit der Einstellung

Ressource	Quota
	Arbeitsspeicher (MB) erhöhen oder verringern. Bei 1 769 MB hat eine Funktion das Äquivalent von einer vCPU.
Funktion Zeitüberschreitung	900 Sekunden (15 Minuten). Für Funktionen mit AWS Lambda Verwaltete Instances, asynchrone Aufrufe und Aufrufe zur Zuordnung von Ereignisquellen (außer Amazon MQ und Amazon DocumentDB) unterstützen einen zulässigen Höchstwert von 5.400 Sekunden (90 Minuten).
Funktion Umgebungsvariablen	4 KB, für alle Umgebungsvariablen, die mit der Funktion verknüpft sind, im Aggregat
Funktion ressourcenbasierte Richtlinie	20 KB
Funktionsebenen	5 Ebenen
Funktion – Limit für Gleichzeitigkeitsskalierung	Für jede Funktion 1 000 Ausführungsumgebungen alle 10 Sekunden
Aufrufnutzlast (Anfrage und Antwort)	Jeweils 6 MB für Anfrage und Antwort (synchron) 200 MB für jeden gestreamte Antwort (synchron) 1 MB (asynchron) 1 MB für die kombinierte Gesamtgröße von Anforderungszeile und Kopfdaten

Ressource	Quota
Bandbreite für gestreamte Antworten	Unbegrenzt für die ersten 6 MB der Antwort Ihrer Funktion Für Antworten, die größer als 6 MB sind, 2 Mbit/s für den Rest der Antwort
Netzwerkbandbreite pro Ausführungsumgebung	625 Mbit/s. Für Funktionen, die nicht an eine VPC gebunden sind, können Sie eine Erhöhung über den Konsole „Dienstkontingente“. Nach der Genehmigung skaliert die Netzwerkbandbreite proportional zum Arbeitsspeicher. Sie beginnt bei 2.048 MB Arbeitsspeicher und erreicht bei 10.240 MB ein Maximum von bis zu 3.000 Mbit/s.
Größe des Bereitstellungspakets (ZIP-Dateiarchiv)	50 MB (gezippt, wenn sie über die Lambda-API oder SDKs hochgeladen werden). Laden Sie größere Dateien mit Amazon S3 hoch. 50 MB (beim Hochladen über die Lambda-Konsole) 250 MB Die maximale Größe des Inhalts eines Bereitstellungspakets, einschließlich Ebenen und benutzerd efinierter Laufzeiten. (ungezippt)
Größe der Container-Image-Einstellungen	16 KB
Codepaketgröße des Container-Images	10 GB (maximale unkomprimierte Image-Größe, einschließlich aller Ebenen)

Ressource	Quota
Testereignisse (Konsoleneditor)	10
`/tmp`-Verzeichnisspeicher	Zwischen 512 MB und 10 240 MB, in 1-MB-Schritten.
Dateibeschreibungen	1,024 **Note** Lambda Managed Instances verwenden ein höheres Limit für Dateideskriptoren von 4.096. Weitere Informati onen finden Sie unter Grundlegendes zur Lambda Managed Instance-Ausführun gsumgebung.
Ausführung processes/threads	1,024 **Note** Lambda Managed Instances verwenden die Standardp rozess- und Thread-Limits von Bottlerocket. Weitere Informationen finden Sie unter Grundlegendes zur Lambda Managed Instance-Ausführungsumgebung.

Lambda-API-Anforderungen

Die folgenden Kontingente sind Lambda-API-Anfragen zugeordnet.

Ressource	Kontingent
Aufrufanfragen pro Funktion pro Region (synchron)	Jede Instance Ihrer Ausführungsumgebung kann bis zu 10 Anfragen pro Sekunde bearbeiten. Mit anderen Worten, das Gesamtauflimit beträgt das 10-fache Ihres Gleichzeitigkeitslimits. Siehe Die Skalierung von Lambda-Ereignisfunktionen verstehen.
Aufrufanfragen pro Funktion pro Region (asynchron)	Jede Instance Ihrer Ausführungsumgebung kann eine unbegrenzte Anzahl an Anfragen bearbeiten. Mit anderen Worten, das Gesamtlimit für Aufrufe basiert nur auf der für Ihre Funktion verfügbaren Gleichzeitigkeit. Siehe Die Skalierung von Lambda-Ereignisfunktionen verstehen.
Aufrufanforderungen pro Funktionsversion oder Alias (Anfragen pro Sekunde)	10 x zugewiesene Provisioned Concurrency **Note** Dieses Kontingent gilt nur für Funktionen, die Provisioned Concurrency verwenden.
GetFunction-API-Anforderungen	100 Anforderungen pro Sekunde. Kann nicht erhöht werden.
GetPolicy-API-Anforderungen	15 Anforderungen pro Sekunde. Kann nicht erhöht werden.

Ressource	Kontingent
Rest der API-Anfragen der Kontrollebene (ohne Aufruf), GetFunction, und GetPolicy Anfragen)	15 Anforderungen pro Sekunde über alle APIs (nicht 15 Anforderungen pro Sekunde pro API). Kann nicht erhöht werden.

Dauerhafte Funktionen

Die folgenden Kontingente gelten für langlebige Funktionen. Die Kontingente gelten pro AWS-Region. Höhere Standardwerte gelten in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland), wie in der Tabelle angegeben. Sofern nicht anders angegeben, können diese Kontingente erhöht werden. Weitere Informationen finden Sie unter Beantragen einer Kontingenterhöhung im Service-Quotas-Benutzerhandbuch.

Ressource	Standardkontingent
Maximal laufende, langlebige Ausführungen	5.000.000 pro Region. 10.000.000 in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland). Dies ist eine weiche Grenze, die erhöht werden kann.
Maximale Anzahl dauerhafter Operationen pro dauerhafter Ausführung	30 000 Kann nicht erhöht werden. Weitere Informationen finden Sie unter Verfügbare langlebige Operationen.
Dauerhafter Ausführungsspeicher, geschrieben in Megabyte	100 MB Kann nicht erhöht werden. Die kumulierte Nutzlastgröße wurde durch dauerhafte Funktionen pro Ausführung beibehalten. Weitere

Ressource	Standardkontingent
	Informationen finden Sie unter persistente Daten pro dauerhaftem Vorgang.
Maximale Nutzlastgröße pro Dauerbetrieb	1 MB Kann nicht erhöht werden. Gilt für Checkpoint- und Callback-Payloads.
Aufrufrate der Funktion zur Initiierung dauerhafter Ausführungen	3.000 Anfragen pro Sekunde. 6.000 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).
CheckpointDurableExecution-API-Anforderungen	10.000 Anfragen pro Sekunde. 20.000 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).
GetDurableExecution-API-Anforderungen	30 Anfragen pro Sekunde.
GetDurableExecutionHistory-API-Anforderungen	15 Anforderungen pro Sekunde.
GetDurableExecutionState-API-Anforderungen	10.000 Anfragen pro Sekunde. 20.000 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).
ListDurableExecutionsByFunction-API-Anforderungen	15 Anforderungen pro Sekunde.
SendDurableExecutionCallbackFailure-API-Anforderungen	3.000 Anfragen pro Sekunde. 6.000 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).

Ressource	Standardkontingent
SendDurableExecutionCallbackHeartbeat-API-Anforderungen	3.000 Anfragen pro Sekunde. 6.000 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).
SendDurableExecutionCallbackSuccess-API-Anforderungen	3.000 Anfragen pro Sekunde. 6.000 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).
StopDurableExecution-API-Anforderungen	300 Anfragen pro Sekunde. 600 Anfragen pro Sekunde in den USA Ost (Nord-Virginia), den USA West (Oregon) und Europa (Irland).

Lambda-MicroVMs

Lambda MicroVMS legt Kontingente für Rechen-, Speicher- und API-Anfragen fest. Als anpassbar markierte Kontingente können über die Service Quota-Konsole erhöht werden.

> **Note**
>
> Lambda-MicroVMs unterstützen den ARM64 (AWS Graviton () -Architektur.

Datenverarbeitung und Speicherung

Ressource	Standardkontingent	Einstellbar
Allen MicroVMs zugewiesener Speicher (pro Konto, pro Region)	400 GB (d. h. 200 microVMs mit jeweils 2 GB konfiguriertem Speicher oder 400	Ja

Ressource	Standardkontingent	Einstellbar
	microVMs mit jeweils 1 GB konfiguriertem Speicher). 1.024 GB in den USA Ost (Nord-Virginia), USA West (Oregon), USA Ost (Ohio) und Asien-Pazifik (Tokio) (d. h. 512 microVMs mit jeweils 2 GB konfiguriertem Speicher oder 1.024 microVMs mit jeweils 1 GB konfiguriertem Speicher). Bis zum Vierfachen dieses Kontingents hochbelastbar.	
Maximale Ausführungsdauer pro microVM	8 Stunden (28.800 Sekunden)	Nein

Bilder und Versionen

Ressource	Standardkontingent	Einstellbar
MicroVM-Images pro Konto pro Region	100	Ja
Versionen pro MicroVM-Image	50	Ja
Gleichzeitige Image-Builds (pro Konto, pro Region)	5 10 in den USA Ost (Nord-Virginia), USA	Ja

Ressource	Standardkontingent	Einstellbar
	West (Oregon), USA Ost (Ohio) und Asien-Pazifik (Tokio)	

Per-MicroVM Durchsatz

Ressource	Limit	Einstellbar
Gleichzeitige Verbindungen pro MicroVM	8 (1 vCPU), 16 (2 vCPU), 32 (4 vCPU), 64 (8 vCPU), 128 (16 vCPU)	Nein
Anfragen pro Sekunde pro MicroVM	40 (4 vCPU/ 8 GB), 160 (16 vCPU/ 32 GB)	Nein

API-Ratenlimits

API-Operation	Rate (TPS)	Burst	Einstellbar
`RunMicrovm`	5	5	Ja
`ResumeMicrovm`	5	5	Ja
`SuspendMicrovm`	2	2	Ja
`TerminateMicrovm`	10	10	Ja
`GetMicrovm`	100	100	Ja
`CreateMicrovmAuthToken`	50	50	Ja
`CreateMicrovmShell AuthToken`	5	5	Ja

> **Note**
>
> TPS = Transaktionen pro Sekunde. Diese Ratenlimits gelten pro Konto und pro Region. Versuchen Sie erneut, gedrosselte Anfragen mit exponentiellem Backoff mit Jitter zu verwenden.

Lambda-Web-Funktionen

Mit AWS Lambda Mit Web Functions führen Sie Webanwendungen und API-Backends als serverlose HTTP-Server aus. Kontingente sind die Höchstwerte für Web Functions-Ressourcen und -Einstellungen. Beispiele hierfür sind die Anzahl der Endpunkte pro Web Function, der Speicher einer Ausführungsumgebung und die Anforderungen pro Sekunde für Ihr Konto. Wenn Sie diese vor der Bereitstellung kennen, können Sie Ihre Anwendung entwerfen und die Skalierungseinstellungen der Endpunkte entsprechend anpassen. Die meisten Kontingente haben Standardwerte, die für typische Workloads geeignet sind und nicht geändert werden können. Sie können Erhöhungen für die anpassbaren Kontokontingente beantragen, die je nach Konto variieren können.

Kontingente für Webfunktionen gelten in den folgenden Bereichen:

- Pro Web-Funktion — Endpunkte und Revisionen.
- Pro Endpunkt — Maximale Umgebungen, Ratenlimit und Anzahl der Revisionen, die den Traffic gemeinsam nutzen.
- Pro Revision — Größe des Bereitstellungspakets, Größe der Umgebungsvariablen und Timeout für Anfragen.
- Pro Ausführungsumgebung — vCPUs, Speicher, gleichzeitige Anfragen, `/tmp` Verzeichnisgröße, Dateideskriptoren, Prozesse und Threads sowie Timeout bei der Initialisierung.
- Pro Anfrage — Header-Größe der Anfrage.
- Pro Konto, in jedem AWS-Region — Anfragen pro Sekunde und Arm64-vCPUs.

> **Note**
>
> Rufen Sie an, um die Kontingentwerte zu sehen, die für Ihr Konto in einer Region gelten `GetWebAccountSettings`. Informationen zur Überwachung der Nutzung

anhand dieser Kontingente finden Sie unter the section called “Regionale Nutzungs- und Drosselungsmetriken”.

Kontingente für Webfunktionen, Revisionen und Ausführungsumgebungen

In der folgenden Tabelle sind die Kontingente für jede Webfunktion, Revision, Ausführungsumgebung und Anforderung aufgeführt.

Ressource	Standardkontingent	Einstellbar
Endpunkte pro Webfunktion	10	Nein
Revisionen pro Webfunktion	50 (automatisches Bereinigungsziel). Die Anzahl kann vorübergehend 50 überschreiten. Siehe the section called “Revisionsgrenzen und automatische Bereinigung”.	Nein
Größe des Bereitstellungspakets	512 MB (entpackt)	Nein
Größe der Umgebungsvariablen	32 KB	Nein
Größe des Headers der Anfrage	128 KiB	Nein
`/tmp` Größe des Verzeichnisses	512 MB	Nein
vCPUs pro Ausführungsumgebung	2	Nein
Speicher pro Ausführungsumgebung	2 GB	Nein
Gleichzeitige Anfragen pro Umgebung (`maxConcurrencyPerEnvironment`)	64 (Bereich 1—128). Siehe the section called “Dynamisches	Nein

Ressource	Standardkontingent	Einstellbar
	Parallelitätslimit pro Umgebung".	
Anforderungstimeout	30 Sekunden (Bereich 3—900)	Nein
Timeout bei der Initialisierung (Zeit, in der Ihr Server mit dem Abhören beginnt)	10 Sekunden	Nein
Dateideskriptoren pro Ausführungsumgebung	1,024	Nein
Prozesse und Threads pro Ausführun gsumgebung	1,024	Nein

Endpunktkontingente

In der folgenden Tabelle sind die Endpunktkontingente aufgeführt.

Ressource	Standardkontingent	Einstellbar
Maximale Anzahl Umgebungen pro Endpunkt (`scalingConfig.maxEnvironments`)	Optional. Standardm äßig nicht festgeleg t. Gültige Werte 2— 10.000. Wird in jeder Region durchgesetzt. Das vCPU-Kontingent des Kontos kann den Endpunkt auf weniger Ausführungsumgebun gen beschränken.	Nein
Ratenlimit pro Endpunkt (`throttleC onfig.rateLimit`)	Optional. Standardm äßig nicht festgeleg t. Gültige Werte sind 0, 100—1.000 in Schritten von 100	Nein

Ressource	Standardkontingent	Einstellbar
	und 2.000—10.000 in Schritten von 1.000 Anfragen pro Sekunde. Lambda setzt dieses Limit in jeder Region separat durch. Ein Wert von 0 blockiert den gesamten Verkehr. Wenn nicht festgeleg t, gilt das Kontingent von 10.000 Anfragen pro Sekunde pro Endpunkt.	
Maximale Revisionen pro Endpunkt (Revision sgewichte)	2	Nein
Gewichtsbereich der Revision	1-100	Nein

Kontenkontingente

In der folgenden Tabelle sind die Kontokontingente aufgeführt, die für jedes Konto gelten AWS-Region.

Ressource	Standardkontingent	Einstellbar
Anfragen pro Sekunde pro Konto	100 000	Ja
Arm64-vCPUs für Ausführungsumgebungen, pro Konto, pro Region	2.000. Jede Ausführungsumgebun g verwendet 2 vCPUs.	Ja

Netzwerkbeschränkungen

Die folgenden Einschränkungen gelten für Web Function-Netzwerke.

Item	Einschränkung
Netzwerkanschlüsse für ausgehenden Datenverkehr pro Revision	0 oder 1
Referenz des Steckverbinders	Version-qualified ARN
Connector-Konto und Region	Entspricht der Web-Funktion
Berechnungsressourcentyp des Connectors	Genau einer; verwenden `WebFunction`
Den Konnektor einer Revision ändern	Eine neue Revision erstellen
Revisionen, die von einem Endpunkt weitergel eitet werden	1 oder 2
Endpunkttypen, die Netzwerkanschlüsse für ausgehenden Datenverkehr unterstützen	Nur `HomeRegion`
Automatische Bereitstellung	`LatestRevision` oder `Disabled` zum `HomeRegion` ; `Disabled` nur für `MultiRegi on` und `PerRegion`
Endpunkttyp und Region festgelegt	Das Problem wurde bei der Erstellung des Endpunk
Direkter öffentlicher Ausgang	Nur `IPv4`
Zugriff auf private Endpunkte mit AWS PrivateLi nk	Nicht verfügbar

Sonstige Services

Kontingente für andere Dienste, wie AWS Identity and Access Management (IAM), Amazon CloudFront (Lambda @Edge) und Amazon Virtual Private Cloud (Amazon VPC) können sich auf Ihre

Lambda-Funktionen auswirken. Weitere Informationen finden Sie unter AWS-Service -Kontingent im Allgemeine Amazon Web Services-Referenz und Lambda mit Ereignissen von anderen aufrufen AWS service.

Viele Anwendungen, an denen Lambda beteiligt ist, verwenden mehrere AWS Dienstleistungen. Da verschiedene Services unterschiedliche Kontingente für verschiedene Funktionen haben, kann es schwierig sein, diese Kontingente für Ihre gesamte Anwendung zu verwalten. API Gateway hat z. B. eine standardmäßige Drosselungsgrenze von 10.000 Anforderungen pro Sekunde, während Lambda eine standardmäßige Gleichzeitigkeitsbeschränkung von 1.000 Anforderungen pro Sekunde hat. Aufgrund dieser Diskrepanz ist es möglich, dass mehr eingehende Anfragen von API Gateway eingehen, als Lambda verarbeiten kann. Sie können dies beheben, indem Sie eine Erhöhung der Lambda-Gleichzeitigkeitsbeschränkung auf das erwartete Datenverkehrsaufkommen beantragen.

Durch Auslastungstests Ihrer Anwendung können Sie die Leistung Ihrer Anwendung von Anfang bis Ende überwachen, bevor Sie sie in der Produktion bereitstellen. Während eines Lasttests können Sie alle Kontingente ermitteln, die einen begrenzenden Faktor für das erwartete Verkehrsaufkommen darstellen und entsprechende Maßnahmen ergreifen.

Dokumentverlauf

In der folgenden Tabelle werden die wichtigen Änderungen am AWS Lambda Entwicklerhandbuch seit Mai 2018 beschrieben. Um Benachrichtigungen über Aktualisierungen dieser Dokumentation zu erhalten, können Sie den RSS-Feed abonnieren.

Änderung	Beschreibung	Datum
AWS verwaltete Aktualisierung der Richtlinien	Lambda hat eine neue AWS verwaltete Richtlinie (`AWSLambdaNetworkConnectorOperatorPolicy`) hinzugefügt, um Berechtigungen zum Erstellen und Verwalten von Elastic Network Interface (ENI) - Ressourcen zu gewähren, die vom Lambda Network Connector verwaltet werden. Weitere Informationen finden Sie unter Lambda-Updates für verwaltete Richtlinien. AWS	19. Juni 2026
AWS verwaltetes Richtlinien-Update	Lambda `AWSLambdaManagedEC2ResourceOperator` hat die verwaltete Richtlinie aktualisiert, um CloudWatch Amazon-Berechtigungen hinzuzufügen (`logs:CreateLogGroup` ,`logs:CreateLogStream` ,`logs:PutLogEvents`). Einzelheiten finden Sie unter Lambda-Aktualisierungen der AWS verwalteten Richtlinien.	10. Juni 2026

Self-managed S3-Code-Speicher	Lambda unterstützt jetzt selbstverwaltete S3-Buckets für die Speicherung von Funktions- und Layer-Code. Sie können Lambda so konfigurieren, dass es direkt von Ihrem S3-Bucket aus auf Ihren Code verweist, wodurch Speicherbeschränkungen entfallen. Lambda-managed Einzelheiten finden Sie unter Self-managed S3-Codespeicher.	30. April 2026
AWS verwaltete Richtlinienaktualisierung	Lambda `AWSLambdaManagedEC2ResourceOperator` hat die verwaltete Richtlinie aktualisiert, um die `ec2:DescribeVpcEncryptionControls` Berechtigung hinzuzufügen. Einzelheiten finden Sie unter Lambda-Aktualisierungen AWS verwalteter Richtlinien.	23. April 2026

AWS verwaltete Richtlinienaktualisierungen	Lambda hat eine neue verwaltete Richtlinie veröffentlicht `AWSLambdaBasicDurableExecutionRolePolicy`, die Schreibberechtigungen für CloudWatch Logs und read/write Berechtigungen für dauerhafte Ausführungs-APIs gewährt, die von den dauerhaften Lambda-Funktionen verwendet werden.	1. Dezember 2025
Service-linked Rolle für Lambda	Lambda unterstützt jetzt eine service-verknüpfte Rolle (`AWSServiceRoleForLambda`), die es Lambda ermöglicht, Instances zu beenden, die als Teil von Lambda-Kapazitätsanbietern verwaltet werden.	30. November 2025
Von Lambda verwaltete Instanzen	Lambda unterstützt jetzt Managed Instances, die dedizierte Rechenkapazität für Ihre Funktionen mit verbesserter Leistung, Sicherheit und Kontrolle bieten. Einzelheiten finden Sie unter Lambda Managed Instances.	30. November 2025
AWS verwaltete Richtlinienaktualisierungen	Lambda hat eine neue AWS verwaltete Richtlinie hinzugefügt (`AWSLambdaBasicDurableExecutionRolePolicy`).	30. November 2025

AWS verwaltete Richtlinienaktualisierungen	Lambda hat die bestehende AWS verwaltete Richtlinie aktualisiert `AWSLambda_FullAccess` (ermöglicht die `iam:CreateServiceLinkedRole` Aktionen `kms:DescribeKey` und).	30. November 2025
AWS verwaltete Aktualisierung der Richtlinien	Lambda hat eine neue AWS verwaltete Richtlinie (`AWSLambdaManagedEC2ResourceOperator`) hinzugefügt, um das automatisierte Amazon EC2-Instance-Management für Lambda-Kapazitätsanbieter zu ermöglichen. Einzelheiten finden Sie unter Lambda-Updates für verwaltete Richtlinien. AWS	30. November 2025
AWS verwaltete Aktualisierung der Richtlinien	Lambda hat eine neue AWS verwaltete Richtlinie hinzugefügt (`AWSLambdaServiceRolePolicy`). Einzelheiten finden Sie unter Lambda-Updates für AWS verwaltete Richtlinien.	30. November 2025
AWS verwaltete Richtlinienaktualisierungen	Lambda hat zwei bestehende AWS verwaltete Richtlinien (`AWSLambda_ReadOnlyAccess` und`AWSLambda_FullAccess`) aktualisiert.	20. Februar 2025

Node.js 22.x-Laufzeit	Lambda unterstützt jetzt Node.js 22 als verwaltete Laufzeit und als Container-Basis-Image (`nodejs22.x`).	22. November 2024
SnapStart Unterstützung für Python und.NET	Lambda SnapStart ist jetzt für von Python und .NET verwaltete Laufzeiten verfügbar, beginnend mit `python3.12` und. `dotnet8`	18. November 2024
Python-3.13-Laufzeit	Lambda unterstützt jetzt Python 3.13 als verwaltete Laufzeit und Container-Basis-Image.	13. November 2024
Vom Kunden verwaltete Verschlüsselung für .zip-Bereitstellungspakete	Lambda unterstützt jetzt die vom AWS KMS Kunden verwaltete Schlüsselverschlüsselung für ZIP-Bereitstellungspakete.	8. November 2024
Unterstützung für neue SnapStart Regionen	Lambda SnapStart ist jetzt in den folgenden Regionen verfügbar: Europa (Spanien), Europa (Zürich), Asien-Pazifik (Melbourne), Asien-Pazifik (Hyderabad) und Naher Osten (VAE).	12. Januar 2024
AWS verwaltete Richtlinienaktualisierungen	Lambda hat eine bestehende AWS verwaltete Richtlinie aktualisiert (`AWSLambdaVPCAccessExecutionRole`).	5. Januar 2024

Python-3.12-Laufzeit	Lambda unterstützt jetzt Python 3.12 als verwaltete Laufzeit und Container-Basis-Image. Weitere Informationen finden Sie unter Python 3.12-Laufzeit, die jetzt im AWS Lambda AWS Compute-Blog verfügbar ist.	14. Dezember 2023
Java-21-Laufzeit	Lambda unterstützt jetzt Java 21 als Image für die verwaltete Laufzeit und Container-Basis-Image (`java21`).	16. November 2023
Node.js 20.x-Laufzeit	Lambda unterstützt jetzt Node.js 20 als verwaltete Laufzeit und als Container-Basis-Image ()`nodejs20.x`) . Weitere Informationen finden Sie unter Node.js 20.x Runtime, jetzt verfügbar AWS Lambda im AWS Compute-Blog.	14. November 2023
Laufzeit „provided.al2023“	Lambda unterstützt jetzt Amazon Linux 2023 als Image für die verwaltete Laufzeit und Container-Basis-Image. Weitere Informationen finden Sie im AWS Compute-Blog unter Einführung der Amazon Linux 2023 Runtime for AWS Lambda.	9. November 2023

IPv6-Unterstützung für Dual-Stack-Subnetze	Lambda unterstützt jetzt den ausgehenden IPv6-Datenverkehr zu Dual-Stack-Subnetzen. Weitere Informationen finden Sie unter IPv6-Unterstützung.	12. Oktober 2023
Serverless-Funktionen und Anwendungen testen	Erfahren Sie mehr über Techniken zum Debuggen und Automatisieren von Tests von Serverless-Funktionen in der Cloud. In den Abschnitten zur Python- und Typescript-Sprache gibt es jetzt ein Kapitel zu Tests sowie Ressourcen. Einzelheiten finden Sie unter Serverless-Funktionen und Anwendungen testen.	16. Juni 2023
Ruby 3.2-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Ruby 3.2. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Ruby.	7. Juni 2023
Antwort-Streaming	Lambda unterstützt jetzt Streaming-Antworten von Funktionen. Weitere Informationen finden Sie unter Konfigurieren einer Lambda-Funktion zum Streamen von Antworten.	06. April 2023

Asynchrone Aufrufmetriken	Lambda veröffentlicht asynchrone Aufrufmetriken. Weitere Informationen finden Sie unter Asynchrone Aufrufmetriken.	9. Februar 2023
Kontrollen der Laufzeitversion	Lambda veröffentlicht neue Laufzeitversionen, die Sicherheitsupdates, Fehlerbehebungen und neue Funktionen enthalten. Sie können jetzt steuern, wann Ihre Funktionen auf die neuen Laufzeitversionen aktualisiert werden sollen. Weitere Informationen finden Sie unter Lambda-Laufzeitaktualisierungen.	23. Januar 2023
Lambda SnapStart	Verwenden Sie Lambda SnapStart , um die Startzeit für Java-Funktionen zu reduzieren, ohne zusätzliche Ressourcen bereitzustellen oder komplexe Leistungsoptimierungen zu implementieren. Weitere Informationen finden Sie unter Verbessern der Startleistung mit Lambda. SnapStart	28. November 2022
Node.js 18-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Node.js 18. Node.js 18 verwendet Amazon Linux 2. Einzelheiten finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	18. November 2022

Lambda: Bedingungsschlüssel SourceFunctionArn	Bei einer AWS Ressource filtert der `lambda:SourceFunctionArn` Bedingungsschlüssel den Zugriff auf die Ressource anhand des ARN einer Lambda-Funktion. Einzelheiten dazu finden Sie unter Arbeiten mit Anmeldeinformationen der Lambda-Ausführungsumgebung.	01. Juli 2022
Node.js 16 Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Node.js 16. Node.js 16 verwendet Amazon Linux 2. Einzelheiten finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	11. Mai 2022
Lambda-Funktions-URLs	Lambda unterstützt jetzt Funktions-URLs, die dedizierte HTTP(S)-Endpunkte für Lambda-Funktionen sind. Details dazu finden Sie unter Lambda-Funktions-URLs.	6. April 2022
Gemeinsame Testereignisse in der Konsole AWS Lambda	Lambda unterstützt jetzt das Teilen von Testereignissen mit anderen Benutzern im selben AWS-Konto. Details finden Sie unter Testen von Lambda-Funktionen mit Hilfe der Konsole.	16. März 2022

PrincipalOrgId in ressourcenbasierten Richtlinien	Lambda unterstützt jetzt das Erteilen von Berechtigungen für eine Organisation in AWS Organizations. Details dazu finden Sie unter Verwenden ressourcenbasierter Richtlinien für AWS Lambda.	11. März 2022
.NET-6-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für .NET 6. Weitere Details finden Sie unter Lambda-Laufzeiten.	23. Februar 2022
Ereignisfilterung für Kinesis-, DynamoDB- und Amazon-SQS-Ereignisquellen	Lambda unterstützt jetzt die Ereignisfilterung für Kinesis-, DynamoDB- und Amazon-SQS-Ereignisquellen. Details dazu finden Sie unter Lambda-Ereignisfilterung.	24. November 2021
mTLS-Authentifizierung für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen	Lambda unterstützt ab sofort die mTLS-Authentifizierung für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MSK.	19. November 2021
Lambda auf Graviton2	Lambda unterstützt jetzt Graviton2 für Funktionen, die arm64-Architektur verwenden. Details dazu finden Sie unter .Lambda-Befehlssatz-Architekturenaus.	29. September 2021

Python-3.9-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Python 3.9. Weitere Details finden Sie unter Lambda-Laufzeiten.	16. August 2021
Neue Laufzeitversionen für Python Node.js und Java	Neue Laufzeitversionen sind für Node.js Python und Java verfügbar. Weitere Details finden Sie unter Lambda-Laufzeiten.	21. Juli 2021
Support für RabbitMQ als Ereignisquelle für Lambda	Lambda bietet jetzt Unterstützung für Amazon MQ für RabbitMQ als Ereignisquelle. Amazon MQ ist ein verwalteter Message-Broker-Service für Apache ActiveMQ und RabbitMQ, der das Einrichten und Betreiben von Message-Brokern in der Cloud vereinfacht. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MQ.	7. Juli 2021

SASL/PLAIN Authentifizierung für selbstverwaltetes Kafka auf Lambda	SASL/PLAIN ist jetzt ein unterstützter Authentifizierungsmechanismus für selbstverwaltete Kafka-Ereignisquellen auf Lambda Kunden, die sie bereits in ihrem selbstverwalteten Kafka-Cluster verwenden SASL/PLAIN , können Lambda jetzt problemlos verwenden, um Verbraucheranwendungen zu erstellen, ohne die Art der Authentifizierung ändern zu müssen. Details dazu finden Sie unter Verwenden von Lambda mit selbstverwaltetem Apache Kafka.	29. Juni 2021
Lambda-Erweiterungs-API	Allgemeine Verfügbarkeit von Lambda-Erweiterungen. Erweiterungen verwenden, um Ihre Lambda-Funktionen zu erweitern. Sie können Erweiterungen verwenden , die von Lambda-Partnern bereitgestellt werden, oder Sie können Ihre eigenen Lambda-Erweiterungen erstellen. Weitere Details finden Sie unter Lambda-Erweiterungs-API.	24. Mai 2021
Neues Lambda-Konsolenerlebnis	Die Lambda-Konsole wurde überarbeitet, um die Leistung und Konsistenz zu verbessern.	2. März 2021

Node.js 14-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Node.js 14. Node.js 14 verwendet Amazon Linux 2. Einzelheiten finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	27. Januar 2021
Lambda-Container-Images	Lambda unterstützt jetzt als Container-Images definierte Funktionen. Sie können die Flexibilität von Container-Tooling mit der Agilität und der betrieblichen Einfachheit von Lambda kombinieren, um Anwendungen zu bauen. Weitere Details finden Sie unter Verwenden von Container-Images mit Lambda.	1. Dezember 2020

Codesignatur für Lambda-Funktionen	Lambda unterstützt jetzt Codesignatur. Administratoren können Lambda-Funktionen so konfigurieren, dass sie bei der Bereitstellung nur signierten Code akzeptieren. Lambda überprüft die Signaturen, um sicherzustellen, dass der Code nicht verändert oder manipuliert wird. Zusätzlich sorgt Lambda dafür, dass der Code von vertrauenswürdigen Entwicklern unterzeichnet wurde, bevor die Bereitstellung zugelassen wird. Weitere Details finden Sie unter Konfigurieren von Codesignatur für Lambda.	23. November 2020
Vorschau: Lambda Runtime Logs API	Lambda unterstützt jetzt die Runtime Logs API. Lambda-Erweiterungen können die Logs-API verwenden, um Protokollstreams in der Ausführungsumgebung zu abonnieren. Weitere Details finden Sie unter Lambda Runtime Logs API.	12. November 2020

Neue Ereignisquelle für Amazon MQ	Lambda bietet jetzt Unterstützung für Amazon MQ als Ereignisquelle. Verwenden Sie eine Lambda-Funktion, um Datensätze von Ihrem Amazon-MQ-Message-Broker zu verarbeiten. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MQ.	5. November 2020
Vorschau: Lambda-Erweiterungs-API	Lambda-Erweiterungen verwenden, um Ihre Lambda-Funktionen zu erweitern. Sie können Erweiterungen verwenden, die von Lambda-Partnern bereitgestellt werden, oder Sie können Ihre eigenen Lambda-Erweiterungen erstellen. Weitere Details finden Sie unter Lambda-Erweiterungs-API.	8. Oktober 2020
Unterstützung für Java 8 und benutzerdefinierte Laufzeiten in AL2	Lambda unterstützt jetzt Java 8 und benutzerdefinierte Laufzeiten unter Amazon Linux 2. Weitere Details finden Sie unter Lambda-Laufzeiten.	12. August 2020

Neue Ereignisquelle für Amazon Managed Streaming for Apache Kafka	Lambda bietet jetzt Unterstützung für Amazon MSK als Ereignisquelle. Verwenden Sie eine Lambda-Funktion mit Amazon MSK, um Datensätze in einem Kafka-Thema zu verarbeiten. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MSK.	11. August 2020
Verwenden von IAM-Bedingungsschlüsseln für Amazon-VPC-Einstellungen	Sie können jetzt Lambda-specific Bedingungsschlüssel für VPC-Einstellungen verwenden. Sie können beispielsweise festlegen, dass alle Funktionen in Ihrer Organisation mit einer VPC sein müssen. Sie können auch die Subnetze und Sicherheitsgruppen angeben, die die Benutzer der Funktion verwenden können bzw. nicht verwenden können. Weitere Details finden Sie unter Konfigurieren einer VPC für IAM-Funktionen.	10. August 2020

Einstellungen für Parallelität für HTTP/2 Kinesis-Stream-Verbraucher	Sie können jetzt die folgenden Parallelitätseinstellungen für Kinesis-Verbraucher mit verbessertem Fan-Out (HTTP/2 Streams) verwenden : ParallelizationFactor,, und MaximumRetryAttemp ts. MaximumRecordAgeIn Seconds DestinationConfig BisectBatchOnFunctionError Einzelheiten finden Sie unter Verwendung AWS Lambda mit Amazon Kinesis.	7. Juli 2020
Batch-Fenster für Kinesis-S tream-Nutzer HTTP/2	Sie können jetzt ein Batch-Fenster (MaximumBatchingWi ndowInSeconds) für HTTP/2 Streams konfigurieren. Lambda liest Datensätz e aus dem Stream, bis es einen vollständigen Batch gesammelt hat oder bis das Batchfenster abläuft. Einzelheiten finden Sie unter Verwenden AWS Lambda mit Amazon Kinesis.	18. Juni 2020
Support für Amazon-EFS-Dateisysteme	Sie können nun ein Amazon-EFS-Dateisystem mit Ihren Lambda-Funktionen für den freigegebenen Netzwerkd ateizugriff verbinden. Weitere Details finden Sie unter Konfigurieren des Dateisyst emzugriffs für Lambda-Fu nktionen.	16. Juni 2020

AWS CDK Beispielanwendungen in der Lambda-Konsole	Die Lambda-Konsole enthält jetzt Beispielanwendungen, die das für verwenden. AWS Cloud Development Kit (AWS CDK) TypeScript Das AWS CDK ist ein Framework, mit dem Sie Ihre Anwendungsressourcen in Python TypeScript, Java oder.NET definieren können.	1. Juni 2020
Unterstützung für .NET Core 3.1.0 Runtime in AWS Lambda	AWS Lambda unterstützt jetzt die .NET Core 3.1.0-Laufzeit. Weitere Details finden Sie unter .NET Core CLI.	31. März 2020
Support für API-Gateway-Metriken für HTTP-APIs	Aktualisierte und erweiterte Dokumentation für die Verwendung von Lambda mit API Gateway, einschließlich Unterstützung für HTTP-APIs. Es wurde eine Beispielanwendung hinzugefügt, die eine API erstellt und mit CloudFormation funktioniert. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon API Gateway.	23. März 2020

Ruby 2.7	Für Ruby 2.7 steht eine neue Laufzeit, ruby2.7, zur Verfügung. Dies ist die erste Ruby-Laufzeitumgebung, die Amazon Linux 2 verwendet . Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Ruby.	19. Februar 2020
Gleichzeitigkeitsmetriken	Lambda meldet nun die Metrik `ConcurrentExecutions` für alle Funktionen, Aliase und Versionen. Sie können ein Diagramm für diese Metrik auf der Überwachungsseite für Ihre Funktion anzeigen. Bisher wurde `ConcurrentExecutions` nur auf Kontoebene und für Funktionen gemeldet, die reservierte Gleichzeitigkeit verwenden . Details hierzu finden Sie unter AWS Lambda -Funktionsmetriken.	18. Februar 2020

Aktualisieren auf Funktionszustände	Funktionszustände werden jetzt standardmäßig für alle Funktionen erzwungen. Wenn Sie eine Funktion mit einer VPC verbinden, erstellt Lambda gemeinsame Elastic-Network-Schnittstellen. Dadurch kann Ihre Funktion skaliert werden, ohne zusätzliche Netzwerkschnittstellen erstellen zu müssen. Während dieser Zeit können Sie keine zusätzlichen Operationen für die Funktion ausführen, einschließlich der Aktualisierung der Konfiguration und der Veröffentlichung von Versionen. In einigen Fällen ist der Aufruf ebenfalls betroffen. Details zum aktuellen Status einer Funktion sind über die Lambda-API verfügbar. Diese Aktualisierung wird phasenweise veröffentlicht. Einzelheiten finden Sie im AWS Compute-Blog unter Aktualisierter Lambda-States-Lebenszyklus für VPC-Netzwerke. Weitere Informationen zu Zuständen finden Sie unter AWS Lambda -Funktionszustände.	24. Januar 2020

Aktualisierungen der API-Ausgabe der Funktionskonfiguration	Für Funktionen, die eine Verbindung zu StateReasonCode einer VPC herstellen, wurden InvalidSubnet, SubnetOutOfIPAddresses Ursachencodes zu LastUpdateStatusReasonCode (InvalidSubnet, InvalidSecurityGroup InvalidSecurityGroup) und (,) hinzugefügt. Weitere Informationen zu Zuständen finden Sie unter AWS Lambda -Funktionszustände.	20. Januar 2020
Bereitgestellte Gleichzeitigkeit	Sie haben nun die Möglichkeit, bereitgestellte Gleichzeitigkeit (Provisioned Concurrency) für eine Funktionsversion oder einen Alias zuzuweisen. Mit der bereitgestellten Gleichzeitigkeit kann eine Funktion ohne Schwankungen der Latenz skalieren. Weitere Details finden Sie unter Verwalten von Gleichzeitigkeit für eine Lambda-Funktion.	03. Dezember 2019

Erstellen eines Datenbank-Proxys	Sie können nun mit der Lambda-Konsole einen Datenbank-Proxy für eine Lambda-Funktion erstellen . Ein Datenbank-Proxy ermöglicht es einer Funktion, hohe Gleichzeitigkeitsstufen ohne überlastende Datenbank verbindungen zu erreichen . Weitere Details finden Sie unter Konfigurieren des Datenbankzugriffs für eine Lambda-Funktion.	03. Dezember 2019
Perzentilunterstützung für die Dauer-Metrik	Sie können nun die Dauer-Metrik basierend auf Perzentil en filtern. Weitere Details finden Sie unter AWS Lambda -Metriken.	26. November 2019
Höhere Gleichzeitigkeit für Stream-Ereignisquellen	Eine neue Option für das DynamoDB-Stream- und Kinesis-Stream-Ereignis quellen-Mapping ermöglich t es Ihnen, von jedem Shard mehrere Batches gleichzeitig zu verarbeiten. Wenn Sie die Anzahl gleichzeitiger Stapel pro Shard erhöhen, kann die Gleichzeitigkeit der Funktion bis zu zehnmal so hoch sein wie die Anzahl der Shards in Ihrem Stream. Weitere Details finden Sie unter Lambda-Er eignisquellen-Zuweisung.	25. November 2019

Funktionszustände	Wenn Sie eine Funktion erstellen oder aktualisieren, wechselt diese in einen ausstehenden Zustand, während Lambda Ressourcen zur Unterstützung bereitstellt. Wenn Sie Ihre Funktion mit einer VPC verbinden, kann Lambda sofort eine gemeinsame Elastic-Network-Schnittstelle erstellen, anstatt Netzwerkschnittstellen zu erstellen, wenn Ihre Funktion aufgerufen wird. Dies führt zu einer besseren Leistung der VPC-connected Funktionen, erfordert jedoch möglicherweise ein Update Ihrer Automatisierung. Details hierzu finden Sie unter AWS Lambda -Funktionszustände.	25. November 2019
Fehlerbehandlungsoptionen für den asynchronen Aufruf	Für den asynchronen Aufruf stehen neue Konfigurationsoptionen zur Verfügung. Sie können Lambda so konfigurieren, dass Wiederholversuche begrenzt und ein maximales Ereignisalter festgelegt werden. Weitere Details finden Sie unter Konfigurieren der Fehlerbehandlung für den asynchronen Aufruf.	25. November 2019

Fehlerbehandlung für Stream-Ereignisquellen	Es sind neue Konfigurationsoptionen für Ereignisquellen-Zuweisungen verfügbar, die aus Streams gelesen werden. Sie können DynamoDB-Stream- und Kinesis-Stream-Ereignisquellen-Zuweisung konfigurieren, um Wiederholversuche zu begrenzen und ein maximales Datensatzalter festzulegen. Wenn Fehler auftreten, können Sie das Ereignisquellen-Mapping so konfigurieren, dass Batches vor dem erneuten Versuch aufgeteilt und Aufrufdatensätze für fehlgeschlagene Batches an eine Warteschlange oder an ein Thema gesendet werden. Weitere Details finden Sie unter Lambda-Ereignisquellen-Mapping.	25. November 2019

Ziele für den asynchronen Aufruf	Sie können Lambda jetzt so konfigurieren, dass Datensätze asynchroner Aufrufe an einen anderen Service gesendet werden. Aufrufdatensätze enthalten Details zu Ereignis, Kontext und Antwort der Funktion. Sie können Aufrufdatensätze an eine SQS-Warteschlange, ein SNS-Thema, eine Lambda-Funktion oder einen Event-Bus senden. EventBridge Weitere Details finden Sie unter Konfigurieren der Ziele für asynchronen Aufruf.	25. November 2019
Neue Laufzeiten für Node.js Python und Java	Neue Laufzeiten sind für Node.js 12, Python 3.8 und Java 11 verfügbar. Weitere Details finden Sie unter Lambda-Laufzeiten.	18. November 2019
FIFO-Warteschlangenunterstützung für Amazon-SQS-Ereignisquellen	Sie können jetzt ein Ereignisquellen-Mapping erstellen, die aus einer FIFO (First-in-First-out)-Warteschlange liest. Bisher wurden nur Standardwarteschlangen unterstützt. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon SQS.	18. November 2019

Erstellen von Anwendungen in der Lambda-Konsole	Die Anwendungserstellung in der Lambda-Konsole ist jetzt allgemein verfügbar. Anweisungen finden Sie unter Anwendungen in der Lambda-Konsole verwalten.	31. Oktober 2019
Erstellen von Anwendungen in der Lambda-Konsole (Beta)	Sie können nun eine Lambda-Anwendung mit einer integrierten fortlaufenden Bereitstellungs-Pipeline in der Lambda-Konsole erstellen. Die Konsole stellt Beispielanwendungen bereit, die Sie als Ausgangspunkt für Ihr eigenes Projekt verwenden können. Wählen Sie zwischen AWS CodeCommit und GitHub für die Quellcodeverwaltung. Jedes Mal, wenn Sie Änderungen zu Ihrem Repository übertragen, wird die enthaltene Pipeline automatisch erstellt und bereitgestellt. Anweisungen finden Sie unter Anwendungen in der Lambda-Konsole verwalten.	3. Oktober 2019

Leistungsverbesserungen für VPC-connected Funktionen	Lambda verwendet jetzt eine neue Art von Elastic-Network-Schnittstelle, die von allen Funktionen in einem Virtual-Private-Cloud-(VPC)-Subnetz gemeinsam genutzt wird. Wenn Sie eine Funktion mit einer VPC verbinden, erstellt Lambda eine Netzwerkschnittstelle für jede Kombination aus Sicherheitsgruppe und Subnetz, die Sie auswählen. Wenn die freigegebenen Netzwerkschnittstellen verfügbar sind, muss die Funktion bei der Skalierung nach oben keine zusätzlichen Netzwerkschnittstellen erstellen. Dies verbessert die Startup-Zeiten erheblich. Weitere Details finden Sie unter Konfigurieren einer Lambda-Funktion für den Zugriff auf Ressourcen in einer VPC.	03. September 2019

Stream-Batch-Einstellungen	Sie können jetzt ein Batch-Fenster für Amazon-DynamoDB- und Amazon-Kinesis-Ereignisquellen-Mappings konfigurieren. Konfigurieren Sie ein Batch-Fenster von bis zu fünf Minuten, um eingehende Datensätze zu puffern, bis ein vollständiger Batch verfügbar ist. Dies reduziert die Anzahl der Aufrufe Ihrer Funktion, wenn der Stream weniger aktiv ist.	29. August 2019
CloudWatch Integration von Logs & Insights	Die Überwachungsseite in der Lambda-Konsole enthält jetzt Berichte von Amazon CloudWatch Logs Insights.	18. Juni 2019
Amazon Linux 2018.03	Die Lambda-Ausführungsumgebung wurde aktualisiert und verwendet jetzt Amazon Linux 2018.03. Weitere Details finden Sie unter Ausführungsumgebung.	21. Mai 2019

Node.js 10	Eine neue Laufzeit ist für Node.js 10 verfügbar, nodejs10.x. Diese Runtime verwendet Node.js 10.15 und wird regelmäßig mit der neuesten Point-Version von 10 aktualisiert. Node.js Node.js 10 ist auch die erste Runtime, die Amazon Linux 2 verwendet . Einzelheiten finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	13. Mai 2019
GetLayerVersionByArn API	Verwenden Sie die GetLayerVersionByArn API, um Layer-Versionsinformationen mit dem Versions-ARN als Eingabe herunterzuladen. Im Vergleich zu GetLayerVersionByArn können Sie den ARN direkt verwenden GetLayerVersion, anstatt ihn zu analysieren, um den Layer-Namen und die Versionsnummer abzurufen.	25. April 2019
Ruby	AWS Lambda unterstützt jetzt Ruby 2.5 mit einer neuen Runtime. Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Ruby.	29. November 2018

Ebenen	Mit Lambda-Ebenen können Sie Bibliotheken, benutzerdefinierte Laufzeiten und andere Abhängigkeiten getrennt von Ihrem Funktionscode verpacken und bereitstellen. Geben Sie Ihre Ebenen für andere Konten oder die ganze Welt frei. Weitere Details finden Sie unter Lambda-Ebenen.	29. November 2018
Benutzerdefinierte Laufzeiten	Erstellen Sie eine benutzerdefinierte Laufzeit zur Ausführung von Lambda-Funktionen in Ihrer bevorzugten Programmiersprache. Weitere Details finden Sie unter Benutzerdefinierte Lambda-Laufzeiten.	29. November 2018
Application-Load-Balancer-Auslöser	Elastic Load Balancing unterstützt nun Lambda-Funktionen als Ziel für Application Load Balancers. Details finden Sie unter Verwenden von Lambda mit Application Load Balancern.	29. November 2018

Verwenden Sie HTTP/2 Kinesis-Stream-Verbraucher als Auslöser	Sie können HTTP/2 Kinesis-Datenstream-Consumer verwenden, um Ereignisse an diese zu senden. AWS Lambda Stream-Nutzer haben für jeden Shard in Ihrem Datenstream einen eigenen Lesedurchsatz, den Sie nutzen, um die Latenz HTTP/2 zu minimieren. Details finden Sie unter Verwenden von Lambda mit Kinesis.	19. November 2018
Python 3.7	AWS Lambda unterstützt jetzt Python 3.7 mit einer neuen Laufzeit. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Python.	19. November 2018
Erhöhung des Nutzlastlimits für asynchronen Funktionsaufruf	Die maximale Nutzlastgröße für asynchrone Aufrufe wurde von 128 KB auf 256 KB erhöht, was der maximale Nachrichtengröße eines Amazon-SNS-Auslösers entspricht. Weitere Details finden Sie unter Lambda-Kontingente.	16. November 2018
AWS GovCloud (US-East) Region	AWS Lambda ist jetzt in der Region AWS GovCloud (US-East) verfügbar.	12. November 2018

Die AWS SAM Themen wurden in ein separates Entwicklerhandbuch verschoben	Eine Reihe von Themen konzentrierte sich auf das Erstellen serverloser Anwendungen mithilfe von AWS Serverless Application Model (AWS SAM). Diese Themen wurden in das AWS Serverless Application Model -Entwicklerhandbuch verschoben.	25. Oktober 2018
Anzeigen von Lambda-Anwendungen in der Konsole	Sie können den Status Ihrer Lambda-Anwendungen auf der Seite Anwendungen in der Lambda-Konsole anzeigen. Diese Seite zeigt den Status des CloudFormation Stacks. Sie enthält Links zu Seiten mit weiteren Informationen zu den Ressourcen im Stack. Sie können auch aggregierte Metriken für die Anwendung anzeigen und benutzerdefinierte Überwachungs-Dashboards erstellen.	11. Oktober 2018
Timeout-Limit für die Funktionsausführung	Für langlaufende Funktionen wurde das maximal konfigurierbare Ausführungs-Timeout von 5 Minuten auf 15 Minuten erhöht. Weitere Details finden Sie unter Lambda-Limits.	10. Oktober 2018

Unterstützung für die PowerShell Kernsprache in AWS Lambda	AWS Lambda unterstützt jetzt die PowerShell Core-Sprache. Weitere Informationen finden Sie unter Programmiermodell für die Erstellung von Lambda-Funktionen in. PowerShell	11. September 2018
Unterstützung für .NET Core 2.1.0 Runtime in AWS Lambda	AWS Lambda unterstützt jetzt die .NET Core 2.1.0-Laufzeit. Weitere Informationen finden Sie unter .NET Core-CLI.	9. Juli 2018
Updates jetzt über RSS verfügbar	Sie können jetzt einen RSS-Feed abonnieren, um Releases für dieses Handbuch nachzufolgen.	5. Juli 2018
Unterstützung für Amazon SQS als Ereignisquelle	AWS Lambda unterstützt jetzt Amazon Simple Queue Service (Amazon SQS) als Ereignisquelle. Weitere Informationen finden Sie unter Aufrufen von Lambda-Funktionen.	28. Juni 2018
China (Ningxia)-Region	AWS Lambda ist jetzt in der Region China (Ningxia) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	28. Juni 2018

Frühere Updates

In der folgenden Tabelle sind die wichtigen Änderungen in jeder Version des AWS Lambda - Entwicklerhandbuchs vor Juni 2018 beschrieben.

Änderung	Beschreibung	Date
Runtime-Unterstützung für Node.js Runtime 8.10	AWS Lambda unterstützt jetzt Node.js Runtime-Version 8.10. Weitere Informationen finden Sie unter Erstellen von Lambda-Eventfunktionen mit Node.js.	2. April 2018
Funktions- und Aliasänderungs-IDs	AWS Lambda unterstützt jetzt Revisions-IDs für Ihre Funktionsversionen und Aliase. Mit diesen IDs können Sie bedingte Aktualisierungen bei der Aktualisierung Ihrer Funktionsversion oder Alias-Ressourcen nachverfolgen und anwenden.	25. Januar 2018
Unterstützung für Go und .NET 2.0 bei Laufzeit	AWS Lambda hat Laufzeitunterstützung für Go und.NET 2.0 hinzugefügt. Weitere Informationen erhalten Sie unter Erstellen von Lambda-Eventfunktionen mit Go und Lambda-Ereignisfunktionen mit C# erstellen.	15. Januar 2018
Umgestaltung der Konsole	AWS Lambda hat eine neue Lambda-Konsole eingeführt, um Ihre Erfahrung zu vereinfachen, und einen Cloud9-Code-Editor hinzugefügt, mit dem Sie Ihren Funktionscode besser debuggen und überarbeiten können.	30. November 2017
Festlegen von Gleichzeitigkeitslimits für einzelne Funktionen	AWS Lambda unterstützt jetzt das Festlegen von Parallelitätsbeschränkungen für einzelne Funktionen. Weitere Informationen finden Sie unter Konfiguration reservierter Parallelität für eine Event-Funktion.	30. November 2017
Verschiebung von Datenverkehr mithilfe von Aliassen	AWS Lambda unterstützt jetzt die Verlagerung des Datenverkehrs mit Aliasen. Weitere Informationen finden Sie unter Erstellen Sie eine fortlaufende Bereitstellung mit gewichteten Aliasnamen.	28. November 2017

Änderung	Beschreibung	Date
Graduelle Code-Bereitstellung	AWS Lambda unterstützt jetzt die sichere Bereitstellung neuer Versionen Ihrer Lambda-Funktion mithilfe von Code Deploy. Weitere Informationen finden Sie unter Graduelle Code-Bereitstellung.	28. November 2017
Region China (Peking)	AWS Lambda ist jetzt in der Region China (Peking) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	9. November 2017
Einführung von SAM Local	AWS Lambda stellt SAM Local (jetzt bekannt als SAM CLI) vor, ein AWS CLI Tool, das Ihnen eine Umgebung bietet, in der Sie Ihre serverlosen Anwendungen lokal entwickeln, testen und analysieren können, bevor Sie sie in die Lambda-Laufzeit hochladen. Weitere Informationen finden Sie unter Testen und Debuggen von Serverless Anwendungen.	11. August 2017
Region Kanada (Zentral)	AWS Lambda ist jetzt in der Region Kanada (Central) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	22. Juni 2017
South America (São Paulo) Region	AWS Lambda ist jetzt in der Region Südamerika (São Paulo) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	6. Juni 2017
AWS Lambda Unterstützung für AWS X-Ray.	Lambda führt Unterstützung für ein X-Ray, mit der Sie Leistungsprobleme mit Ihren Lambda-Anwendungen erkennen, analysieren und optimieren können. Weitere Informationen finden Sie unter Visualisieren Sie Aufrufe von Lambda-Ereignisfunktionen mit AWS X-Ray.	19. April 2017

Änderung	Beschreibung	Date
Region Asien-Pazifik (Mumbai)	AWS Lambda ist jetzt in der Region Asien-Pazifik (Mumbai) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	28. März 2017
AWS Lambda unterstützt jetzt Node.js Runtime v6.10	AWS Lambda Unterstützung für Node.js Runtime v6.10 hinzugefügt. Weitere Informationen finden Sie unter Erstellen von Lambda-Eventfunktionen mit Node.js.	22. März 2017
Region Europa (London)	AWS Lambda ist jetzt in der Region Europa (London) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	1. Februar 2017
AWS Lambda Unterstützung für die.NET-Runtime, Lambda @Edge (Preview), Dead Letter Queues und die automatis che Bereitste llung serverloser Anwendungen.	AWS Lambda Unterstützung für C# hinzugefügt. Weitere Informationen finden Sie unter Lambda-Ereignisfunktionen mit C# erstellen. Lambda @Edge ermöglicht es Ihnen, Lambda-Funktionen an den AWS Edge-Standorten als Reaktion auf Ereignisse auszuführen. CloudFront Weitere Informationen finden Sie unter Anpassen am Edge mit Lambda @Edge.	3. Dezember 2016
AWS Lambda fügt Amazon Lex als unterstützte Ereignisq uelle hinzu.	Mit Lambda und Amazon Lex können Sie Chatbots für verschiedene Services wie Slack und Facebook schnell erstellen.	30. November 2016
US West (N. Californi a) Region	AWS Lambda ist jetzt in der Region USA West (Nordkali fornien) verfügbar. Weitere Informationen zu Lambda-Re gionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	21. November 2016

Änderung	Beschreibung	Date
Es wurde das AWS SAM für die Erstellung und Bereitstellung von Lambda-based Anwendungen und die Verwendung von Umgebungsvariablen für Lambda-Funktionskonfigurationseinstellungen eingeführt.	AWS SAM: Sie können jetzt die verwenden AWS SAM , um die Syntax für das Ausdrücken von Ressourcen in einer serverlosen Anwendung zu definieren. Um Ihre Anwendung bereitzustellen, geben Sie einfach die Ressourcen, die Sie als Teil Ihrer Anwendung benötigen, zusammen mit den zugehörigen Berechtigungsrichtlinien in einer AWS CloudFormation Vorlagendatei (entweder in JSON oder YAML geschrieben) an, verpacken Sie Ihre Bereitstellungsartefakte und stellen Sie die Vorlage bereit. Umgebungsvariablen: Sie können Umgebungsvariablen verwenden, um Konfigurationseinstellungen für Ihre Lambda-Funktion vorzunehmen, ohne dabei den Code für die Funktion ändern zu müssen. Weitere Informationen finden Sie unter Arbeiten mit Lambda-Umgebungsvariablen.	18. November 2016
Region Asien-Pazifik (Seoul)	AWS Lambda ist jetzt in der Region Asien-Pazifik (Seoul) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	29. August 2016
Asia Pacific (Sydney) Region	Lambda ist jetzt in der Region Asien-Pazifik (Sydney) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	23. Juni 2016
Aktualisierungen für die Lambda-Konsole	Die Lambda-Konsole wurde aktualisiert, um das Verfahren für die Erstellung von Rollen zu vereinfachen.	23. Juni 2016
AWS Lambda unterstützt jetzt Node.js Runtime v4.3	AWS Lambda Unterstützung für Node.js Runtime v4.3 hinzugefügt. Weitere Informationen finden Sie unter Erstellen von Lambda-Eventfunktionen mit Node.js.	07. April 2016

Änderung	Beschreibung	Date
Region Europa (Frankfurt)	Lambda ist jetzt in der Region Europa (Frankfurt) verfügbar . Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	14. März 2016
VPC-Unterstützung	Sie können eine Lambda-Funktion jetzt für den Zugriff auf Ressourcen in Ihrer VPC konfigurieren. Weitere Informati onen finden Sie unter Lambda Event Functions Zugriff auf Ressourcen in einer Amazon VPC gewähren.	11. Februar 2016
Die Lambda-La ufzeitumgebung wurde aktualisiert.	Die Ausführungsumgebung wurde aktualisiert.	4. November 2015

Änderung	Beschreibung	Date
Unterstützung der Versioning, Python für die Entwicklung von Code für Lambda-Funktionen, geplante Ereignisse und Erhöhung der Ausführungszeit	Sie können den Code für Ihre Lambda-Funktionen jetzt mit Python entwickeln. Weitere Informationen finden Sie unter Lambda-Ereignisfunktionen mit Python erstellen. Versioning: Sie können eine oder mehrere Versionen einer Lambda-Funktion pflegen. Mit der Versioning können Sie steuern, welche Versionen einer Lambda-Funktion in verschiedenen Umgebungen ausgeführt werden (z. B. in einer Entwicklungs-, Test- oder produktiven Umgebung). Weitere Informationen finden Sie unter Versionen der Lambda-Ereignisfunktion verwalten. Geplante Ereignisse: Außerdem können Sie Lambda mithilfe der Lambda-Konsole so einrichten, dass Ihr Code regelmäßig und nach Plan abgerufen wird. Sie können einen festen Takt für die Ausführung (eine Anzahl von Stunden, Tagen oder Wochen) oder einen Cron-Ausdruck angeben. Weitere Informationen finden Sie unter Rufen Sie eine Lambda-Ereignisfunktion nach einem Zeitplan auf. Erhöhung der Ausführungszeit: Sie können Ihre Lambda-Funktion jetzt so einrichten, dass sie bis zu fünf Minuten lang ausgeführt wird; dadurch werden Funktionen mit längeren Laufzeiten ermöglicht, z. B. Aufträge für die Erfassung oder Verarbeitung großer Datenvolumen.	08. Oktober 2015

Änderung	Beschreibung	Date
Unterstützung für DynamoDB Streams	DynamoDB Streams ist jetzt allgemein verfügbar; Sie können die Funktion in allen Regionen verwenden, in denen DynamoDB verfügbar ist. Sie können DynamoDB Streams für Ihre Tabelle aktivieren und eine Lambda-Funktion als Auslöser für die Tabelle verwenden. Auslöser sind benutzerdefinierte Aktionen, die als Reaktion auf Aktualisierungen der DynamoDB-Tabelle ausgeführt werden. Ein Beispiel-Walkthrough finden Sie unte Tutorial: Verwenden AWS Lambda mit Amazon DynamoDB DynamoDB-Streams .	14. Juli 2015
Lambda unterstützt jetzt das Aufrufen von Lambda-Funktionen mit Clients. REST-compatible	Bisher benötigten Sie die AWS SDKs (z. B. SDK für Java, SDK für Android oder AWS SDK für iOS), um Ihre Lambda-Funktion von Ihrer Web-, Mobil- oder IoT-Anwendung aus aufzurufen. AWS AWS Lambda unterstützt jetzt das Aufrufen einer Lambda-Funktion bei REST-compatible Kunden über eine benutzerdefinierte API, die Sie mit Amazon API Gateway erstellen können. Sie können Anfragen an die Endpunkt-URL Ihrer Lambda-Funktion senden. Sie können die Sicherheit auf dem Endpunkt so konfigurieren, dass sie offenen Zugriff ermöglicht, AWS Identity and Access Management (IAM) zur Autorisierung des Zugriffs verwenden oder API-Schlüssel verwenden, um den Zugriff anderer auf Ihre Lambda-Funktionen zu kontrollieren. Ein Beispielübung mit ersten Schritten finden Sie unter Aufrufen einer Lambda-Ereignisfunktion mithilfe eines Amazon API Gateway Gateway-Endpunkts.	09. Juli 2015

Änderung	Beschreibung	Date
In der Lambda-Konsole werden jetzt Vorlagen bereitgestellt, mit denen Sie Lambda-Funktionen einfach erstellen und testen können.	In der Lambda-Konsole wird eine Reihe von Vorlagen bereitgestellt. Jeder Blueprint enthält eine Beispielkonfiguration der Ereignisquelle und einen Beispielcode für Ihre Lambda-Funktion, mit denen Sie problemlos Anwendungen erstellen können. Lambda-based Die Vorlagen sind jetzt in allen Lambda-Übungen mit ersten Schritten enthalten.	09. Juli 2015
Lambda unterstützt jetzt Lambda-Funktionen, die in Java geschrieben wurden.	Sie können Lambda-Code jetzt in Java verfassen. Weitere Informationen finden Sie unter Lambda-Ereignisfunktionen mit Java erstellen.	15. Juni 2015
Lambda unterstützt bei der Erstellung oder Aktualisierung einer Lambda-Funktion jetzt die Angabe eines Amazon-S3-Objekts als ZIP-Datei.	Sie können das Bereitstellungspaket für eine Lambda-Funktion (ZIP-Datei) in einen Amazon-S3-Bucket in derselben Region uploaden, in der Sie die Lambda-Funktion erstellen möchten. Danach können Sie bei der Erstellung oder Aktualisierung der Lambda-Funktion den Bucket-Namen und Objektschlüsselnamen angeben.	28. Mai 2015

Änderung	Beschreibung	Date
In Lambda wurde die allgemeine Unterstützung von mobilen Backends hinzugefügt.	Lambda ist jetzt allgemein für den produktiven Einsatz verfügbar. In dieser Version werden außerdem neue Funktionen eingeführt, mit denen die Erstellung von Backends für Mobiltelefone, Tablets und das Internet of Things (IoT) vereinfacht wird; diese können mit Lambda automatisch skaliert werden, ohne dass eine Infrastruktur bereitgestellt oder verwaltet werden muss. Lambda unterstützt jetzt sowohl synchrone (in Echtzeit) als auch asynchrone Ereignisse. Es sind zusätzliche Funktionen enthalten, u. a. für eine einfachere Konfiguration und Verwaltung von Ereignisquellen. Das Berechtigungsmodell und das Programmiermodell wurden durch die Einführung von Ressourcenrichtlinien für Lambda-Funktionen vereinfacht.	9. April 2015
Vorversion	Vorversion des AWS Lambda -Entwicklerhandbuchs.	13. November 2014

Die vorliegende Übersetzung wurde maschinell erstellt. Im Falle eines Konflikts oder eines Widerspruchs zwischen dieser übersetzten Fassung und der englischen Fassung (einschließlich infolge von Verzögerungen bei der Übersetzung) ist die englische Fassung maßgeblich.